Automated proof-producing abstraction of C code

David Greenaway

School of Computer Science and Engineering

University of New South Wales
Sydney, Australia

Submitted in fulfilment of the requirements for the degree of Doctor of Philosophy

August 2014

Abstract

Before software can be formally reasoned about, it must first be represented in some form of logic. There are two approaches to carrying out this translation: the first is to generate an idealised representation of the program, convenient for reasoning about. The second, safer approach is to perform a precise, conservative translation, at the cost of burdening verification efforts with low-level implementation details.

In this thesis, we present methods for bridging the gap between these two approaches. In particular, we describe algorithms for automatically abstracting low-level C code semantics into a higher level representation. These translations include simplifying program control flow, converting finite machine arithmetic into idealised integers, and translating the byte-level C memory model to a split heap model. The generated abstractions are easier to reason about than the input representations, which in turn increases the productivity of formal verification techniques. Critically, we guarantee soundness by automatically generating proofs that our abstractions are correct. Previous work carrying out such transformations has either done so using unverified translations, or required significant manual proof engineering effort.

Our algorithms are implemented in a new tool named AutoCorres, built on the Isabelle/HOL interactive theorem prover. We demonstrate the effectiveness of our abstractions in a number of case studies, and show the scalability of AutoCorres by translating real-world programs consisting of tens of thousands of lines of code. While our work focuses on a subset of the C programming language, we believe most of our algorithms are also applicable to other imperative languages, such as Java or C#.

Publication List

This thesis is partly based on work described in the following publications:

- D. Greenaway, J. Andronick and G. Klein. 'Bridging the Gap: Automatic Verified Abstraction of C'. In: *Proceedings of the 3rd International Conference on Interactive Theorem Proving*. Volume 7406. LNCS. 2012, pages 99–115. doi: 10.1007/978-3-642-32347-8_8.
- D. Greenaway, J. Lim, J. Andronick and G. Klein. 'Don't Sweat the Small Stuff: Formal Verification of C Code Without the Pain'. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2014, pages 429–439. doi: 10.1145/2594291.2594296.

Acknowledgements

I owe many thanks to my wonderful supervisors Gerwin Klein, June Andronick, and Kevin Elphinstone, for their wise advice, the hours willingly spent poring over my poorly written drafts, and their timely words of encouragement.

I would also like to thank all the people—past and present—in the NICTA Trustworthy Systems group. You ensured that my PhD was never a lonely experience, but provided a constant source of laughter, ideas, and high quality coffee beans. I am particularly indebted to Andrew Boyton (a constant encouragement); Matthew Fernandez, Corey Lewis, and Rohan Ben Jacob Rao (my willing lab rats); Peter Gammie and Toby Murray (for their wise counsel); Japheth Lim (a partner in crime); Thomas Sewell (my oracle for all things Isabelle); and the countless other people who have helped bounce ideas and point me in the right direction.

I would also like to thank Lars Noschinski and Christine Rizkallah for their valuable feedback and boundless patience while using early versions of AutoCorres.

Finally, I would like to thank Hui Yee Greenaway—not only for her long hours of wading through the terrible prose in this thesis—but for her love, encouragement and support; and Zoë Greenaway, for reminding me of what is *actually* important.

Contents

1 Introduction 1

1.1 From source code to logic **1**

1.2 Thesis objectives and contributions **3**

- *Summary of thesis contributions*

1.3 Document overview **5**

2 Related work 9

2.1 C verification **9**

- *Automatic verification of C*
- *Semi-automatic verification of C*
- *Interactive verification of C*

2.2 Abstraction of low-level semantics **14**

2.3 Summary **16**

3 Background 17

3.1 The C programming language **17**

- *Features of C*
- *Undefined and implementation-defined behaviour*

3.2 The Isabelle/HOL theorem prover **22**

- *Interacting with Isabelle*
- *Isabelle's meta-logic*
- *Notation*

3.3 The Simpl language **26**

3.4 Translating C into Isabelle/HOL **30**

- *Translation overview*
- *Converting C types to Isabelle/HOL types*
- *Generation of state types*
- *Generation of Simpl*

3.5 Summary **36**

4 From deep to shallow embeddings 37

4.1 Reasoning in deep and shallow embeddings **37**

4.2 Cock et al.'s monadic framework **39**

- *Introducing the state monad*

- *The state monad*
- *Reasoning about the state monad*
- *Modelling abrupt termination*
- *Reasoning about the exception monad*

4.3 Monadic loops **48**
- *Reasoning about the while-loop combinator*

4.4 Converting Simpl to a monadic representation **58**
- *Proving conversion*
- *Function calls and recursion*

4.5 Structural simplifications of monadic programs **69**
- *Peephole optimisations*
- *Exception elimination*

4.6 Related work **75**

4.7 Conclusion **75**

4.8 Summary **76**

5 Local variable lifting 77

5.1 Lifting local variables out of the program's state **77**
- *Analysing existing local variable usage*
- *Utilising monadic return values*
- *Generating an L2 specification*
- *Proving correspondence between L1 and L2*
- *Proving the L2 specification*

5.2 Further program optimisations **94**

5.3 Type strengthening **95**

5.4 Polishing and final theorem **100**

5.5 Conclusion **102**

5.6 Summary **103**

6 Word abstraction 105

6.1 Reasoning about word arithmetic **105**

6.2 Word abstraction **107**

6.3 Performing the abstraction **108**
- *High-level overview*
- *Implementation in Isabelle/HOL*

6.4 Word abstraction examples **114**
- *Maximum of two integers*
- *Absolute value*
- *Primality testing*

6.5 Extending the rule set **122**

6.6 Related work **123**
6.7 Conclusion **124**
6.8 Summary **124**

7 Heap abstraction 125
7.1 Byte-level versus typed heap reasoning **125**
• *Norrish and Tuch byte-level heap implementation*
• *Working with a byte-level heap*
7.2 Lifting the heap **131**
• *Annotating the heap*
• *Lifting the heap*
• *Limitations of the heap lifting approach*
7.3 Automated state abstraction **138**
• *Generating the abstract state type*
• *Ingredients for generating the abstract program*
• *Heap abstraction ruleset*
• *Example:* `swap`
• *Example: Unsuccessfully abstracting a type-unsafe function*
7.4 Abstracting C structures **149**
• *Example: Suzuki's challenge*
• *Example: in-place reversal of a linked list*
7.5 Mixing low-level and high-level code **153**
• *Example:* `memset`
7.6 Related work **158**
7.7 Conclusion **159**
7.8 Summary **160**

8 Evaluation and experience 161
8.1 High-level reasoning with AutoCorres **162**
• *In-place list reversal*
• *Schorr-Waite algorithm*
8.2 Automatic abstraction in the large **168**
• *Summary and statistics of projects using AutoCorres*
8.3 Conclusion **174**
8.4 Summary **175**

9 Conclusion 177
9.1 Summary **177**
9.2 Thesis contributions **178**
9.3 Applicability to other languages **179**

9.4 Trusting the C-to-Isabelle Parser **179**
9.5 The Simpl language as an input **181**
9.6 Output stability **182**
9.7 Future work **183**
- *Improving performance*
- *Implementing abstract interpretation*
- *Data structure abstractions*
- *An extended C subset*

9.8 Final words **186**

A Appendices 189
A.1 Big-step semantics of Simpl **189**
A.2 Termination of Simpl programs **191**

1 Introduction

Before we can formally verify a computer program, we need three key ingredients.

First, we need a *program to verify*. While this may seem obvious, many practitioners of formal verification often overlook this first ingredient, reasoning instead about high-level models of their programs. Real guarantees about the behaviour of software can only ever be achieved, however, by analysing a concrete implementation.

Second, we need a *reasoning tool*. In the early days of formal verification, a sharp pencil and a clean sheet of paper were the reasoning tools of choice. In modern times, it is more common to use specially developed software—such as interactive theorem provers, SMT solvers, or custom-designed analysis tools—to assist with the tedious and error-prone proofs that emerge during verification.

Finally, before we can start formal verification, we need a *method to translate* the program we wish to verify into the logic of our reasoning tool. For industrial programming languages such as C, Java, or C#, the rigorous approach to carrying out this translation will involve poring over the language's specification—typically a long, dry book written in semi-formal English prose—and then attempting to faithfully encode the program being analysed into a mathematical representation.

This third oft-neglected step must be carried out by *every* tool claiming to perform formal verification. As we shall see below, some tools carry out the step implicitly, never actually revealing the internal mathematical representation to a human; other tools are more explicit about how this process takes place, providing the user with the mathematical interpretation as a starting point for their work.

There are two broad approaches as to how the translation from source code to logic takes place: the first is to create logical representations that are *convenient* to reason about; the second is to create logical representations that are *safe*.

For example, given the simple C program which calculates the maximum of two numbers

```
int max(int a, int b) {
    if (a <= b)
        return b;
    return a;
}
```

One possible *convenient* representation of this program in logic is

$$\begin{aligned}&\mathsf{max}\ a\ b \equiv \\ &\qquad \mathsf{if}\ a \leq b\ \mathsf{then}\ b\ \mathsf{else}\ a\end{aligned}$$

Such a logical representation strips away all the clutter of C's syntax, leaving just its distilled high-level semantics. Both humans and mechanised reasoning tools benefit from clear representations such as this: humans are able to concentrate on the high-level details of the program without distraction, while machines also benefit from not having to wade through gritty details of the language.

While *reasoning* about such a representation of the input C program is a pleasant experience, the risk is that the generated logical representation doesn't match the realities of the concrete program. There are two main ways that such a translation could go wrong. The first is by using mathematical abstractions that don't match reality. A particularly egregious example would be using infinite integers to model 32-bit machine words; while this would provide a significant productivity boost to verification engineers, any 'theorems' proven about the program would cease to be true the moment an integer overflow inadvertently occurred.

The second way that convenient translations can go wrong is more subtle: even when the abstractions used to represent the program are sound, convenient translations are still hard to generate correctly. Control flow must be analysed; the liveness of local variables must be calculated; arithmetic must be scrutinized for overflow and underflow; heap operations must be safely abstracted, and so on. The risk is that if any one of these phases goes wrong—perhaps the tool has a slight implementation error, or was a little optimistic in what it considered a safe abstraction—the generated logical representation may end up bearing no relation to the input program. Any proof on such a model will have no guarantee of actually holding in reality.

The alternative approach to translation—generating *safe* representations—attempts to be more conservative. Instead of trying to be clever in the translation process, the source language is painstakingly modelled to ensure its semantics are precisely captured. The translation from source code to logic is now straightforward, each concept in the source language being directly translated into the equivalent concept in the target logic.

Figure 1.1 gives an example of the `max` function above translated using Norrish's *C-to-Isabelle parser*[1] [84, 85], which we use in this work. Norrish's C-to-Isabelle parser aims to be conservative. This conservatism is clearly visible in its output, with the

[1] The name *C-to-Isabelle parser* is a little misleading, as it not only parses C, but also translates it into Isabelle/HOL. The tool would perhaps be better called a C-to-Isabelle *translator*. In this work, we nevertheless continue to use the name of the tool.

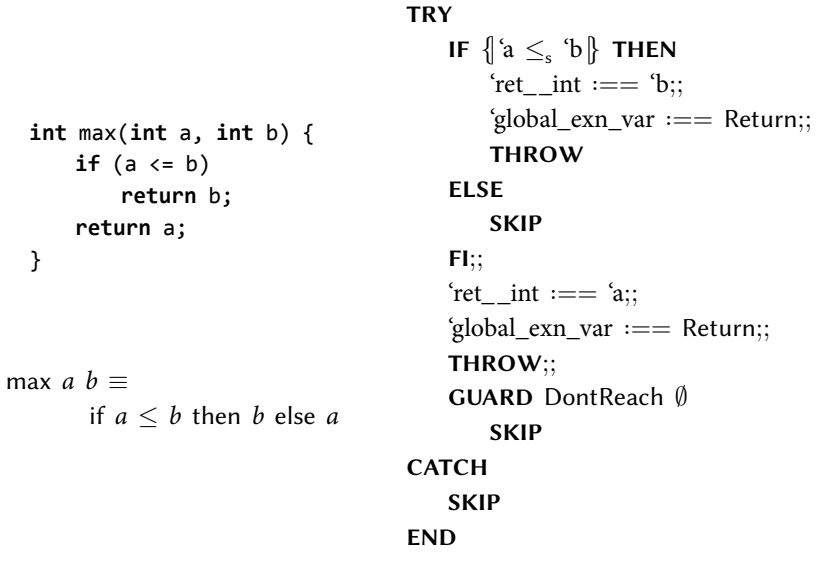


Figure 1.1: The C `max` function, its conservative translation into Isabelle/HOL using Norrish's C-to-Isabelle parser [85], and an idealised logical representation of the function. *(Clockwise from top-left.)*

high-level structure of the program lost in an abundance of intricate detail. For example, the C `return` statement is logically encoded as exceptional control flow; a check to ensure that the function returns a value, instead of dropping off the bottom of the function, is explicitly encoded; the finite word arithmetic implemented by the machine is precisely modelled.

But this conservatism comes at a cost. The tool's output is a mess to look at. As we shall see in later chapters, it's also a mess to work with. But it is, at the very least, a *safe* mess.

1.2 Thesis objectives and contributions

The goal of this thesis is to present a way to bridge the gap between *safe* translations and *convenient* translations. Our process of achieving this is as follows:

- We start by importing programs written in a subset of the C programming language [55] into the Isabelle/HOL interactive theorem prover [83] using Norrish's conservative C-to-Isabelle parser [84, 85];
- Next, we automatically *abstract* the low-level representation of C generated by Norrish's parser into a higher level representation. Our abstraction process (*i*) converts from a deeply embedded representation to a shallowly embedded monadic representation; (*ii*) simplifies control flow of the program; (*iii*) soundly abstracts arithmetic on machine words into arithmetic on unbounded integers and naturals; (*iv*) soundly abstracts the C byte-level heap into a Burstall-Bornat style split-heap; and (*v*) selects an appropriate type for the final version of the specification. Our

generated specification is designed with manual reasoning in mind, but is also suitable for automated reasoning,

- Critically, our abstraction process also generates an LCF-style *proof of correctness*. In particular, we automatically prove that the original low-level input is a refinement of our generated abstract specification. This means that neither our abstraction algorithms nor their implementation need to be trusted by the end user.
- Finally, the user is presented with both a higher level representation of their input program and an Isabelle/HOL theorem providing a formal connection between the output higher level specification and the low-level input program.

The challenge of our work is discovering how we can automatically generate *provably correct* abstractions of low-level imperative programs that are suitable for *human* reasoning, while simultaneously remaining *general* enough that any reasonable property—such as partial correctness, total correctness, full functional verification, and so on—can be proven about the result.

The algorithms and formalisms described in this work have been implemented in a new tool named *AutoCorres* [48], which is implemented in Standard ML and the Isabelle/HOL interactive theorem prover. Using AutoCorres, we can automatically abstract the output of Norrish's C-to-Isabelle parser shown in Figure 1.1 into its representation, also shown in Figure 1.1, automatically generating a proof of correctness in Isabelle/HOL showing that the translation is correct. While AutoCorres only supports a subset of the C language (described in Section 3.4), it is capable of successfully translating several large real-world programs, including an operating system kernel, various graph checking algorithms, a memory allocator, and a real-time operating system.

While our work focuses on the C programming language, we believe that many of the algorithms and formalisms presented in this document are also applicable to other imperative programming languages, such as C#, Java, or SPARK Ada.

1.2.1 Summary of thesis contributions

In summary, the primary contributions of this thesis are as follows:

- We demonstrate how deeply embedded representations of real-world imperative programs may be automatically and verifiably translated into convenient, human-readable, shallowly embedded monadic representations (Chapter 4 and Chapter 5);
- We offer practical optimisations that can be applied to such representations, such as peephole optimisations (Section 4.5.1), exception elimination (Section 4.5.2), flow-sensitive optimisations (Section 5.2), and type strengthening (Section 5.3). These optimisations are designed in such a way that the final output specification remains readable by humans;
- We develop logical frameworks and algorithms to automatically and verifiably rewrite programs using word-based arithmetic into programs that instead operate

on unbounded arithmetic (Chapter 6). We additionally provide case studies showing how the abstracted programs significantly simplify reasoning about programs that use arithmetic (Section 6.4);

- We develop logical frameworks and algorithms to automatically and verifiably rewrite programs using a byte-level heap into programs that instead operate on a Burstall-Bornat style split-heap (Chapter 7). We additionally show how functions abstracted in such a way can interact soundly with functions that need to operate on a byte-level heap (Section 7.5); and finally,
- We evaluate the above-described methods by implementing them in a tool *AutoCorres*, and showing that existing high-level proofs can be applied to the output of AutoCorres with minimal effort (Chapter 8). Additionally, we provide both a qualitative and quantitative analysis of AutoCorres' use in larger projects, both internal and external to our research group (Section 8.2).

AutoCorres is written in the Standard ML programming language, and interacts with Isabelle/HOL through its ML API. While the ML source code of AutoCorres is not verified, all stages of AutoCorres generate a proof in Isabelle/HOL showing that its conversion is correct.

All of the code described in this document is freely available at [48], under an open-source license. This includes the full, machine-checked proofs of the theorems listed this document, the implementation of AutoCorres, and the larger examples and case-studies presented in this thesis.

1.3 Document overview

The remainder of this document describes our specification abstraction techniques, and their implementation in the AutoCorres tool. In particular:

Related work In Chapter 2 we describe existing work that attempts to ease C verification and other approaches to automatically abstracting low-level program semantics into higher level representations.

Background In Chapter 3 we present the work that AutoCorres builds upon. In particular, we give a brief introduction to the C programming language [55] and some of the difficulties it presents. We introduce the Isabelle/HOL interactive theorem prover [83], which incorporates an LCF-style proof kernel that we use to ensure our reasoning is sound. Finally, we describe Norrish's C-to-Isabelle parser [84, 85], which translates C into the Isabelle/HOL-based *Simpl* language [92, 93].

Deep to shallow embeddings Chapter 4 presents a monadic framework developed by Cock et al. [30] that allows shallowly embedded imperative programs to be modelled in Isabelle/HOL. We extend Cock et al.'s work to add support for modelling (potentially non-terminating) imperative-style loops, and present rules to ease reasoning about such loops.

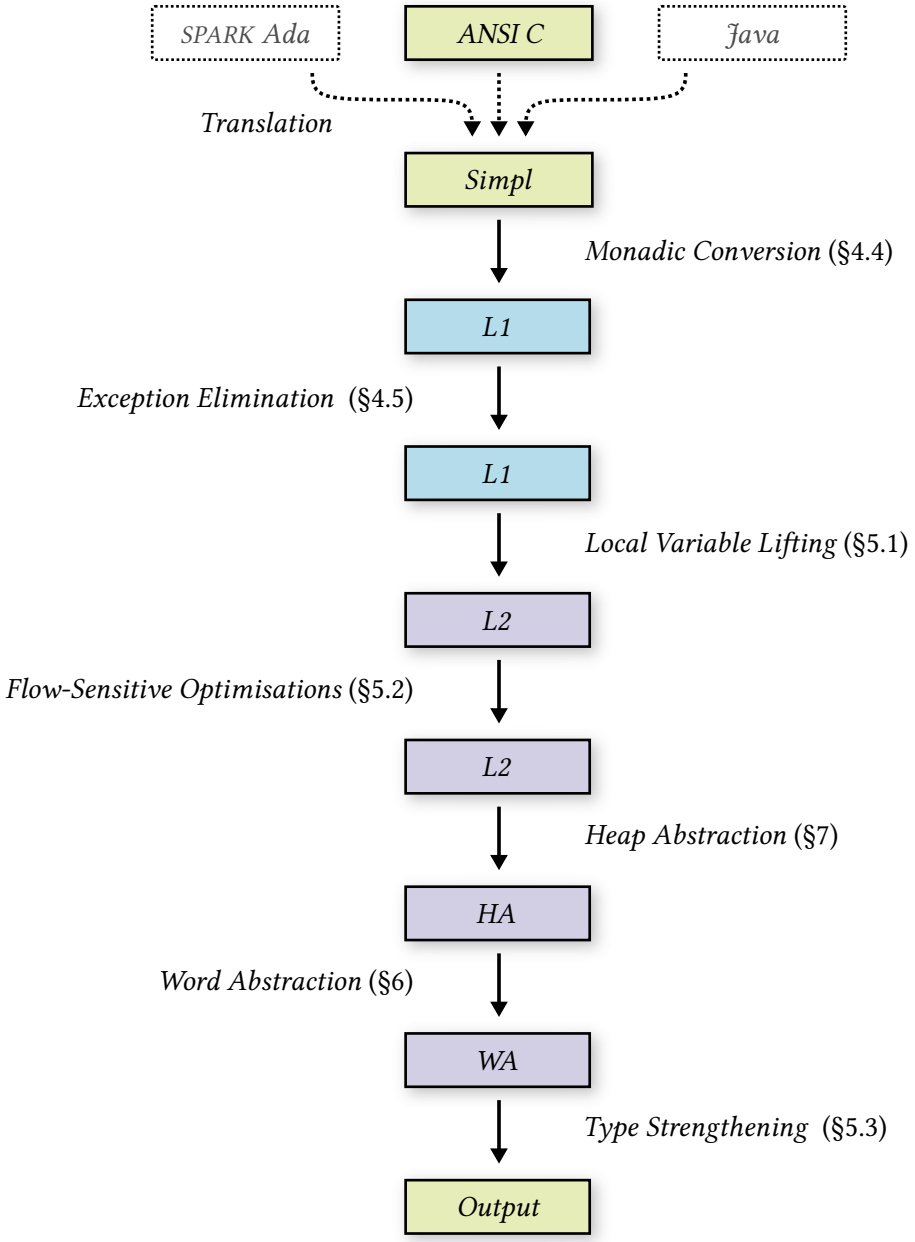


Figure 1.2: Overview of the transformations carried out by AutoCorres, and the locations of their descriptions in this document. Dashed lines indicate translations that need to be trusted, while solid lines indicate transformations that generate proofs of correctness in Isabelle/HOL.

We show how the deeply embedded Simpl language (generated by the C-to-Isabelle parser) can be automatically and provably translated into a monadic representation. We show how this new monadic representation facilitates simplification of the program's representation and, in particular, show how we can significantly simplify the control flow of programs to avoid using exceptions in most cases.

Local variable lifting In Chapter 5 we show how local variables (previously modelled as being stored in the programs state), can be soundly lifted to monadic *bound variables*. This new representation allows us to carry out further simplifications of the resulting specification that take into account the values of local variables.

We next present a technique called *type strengthening* which converts program specifications into a more specialised type, allowing simple programs to be encoded using simple representations. We finally present some examples of reasoning about C programs using the output of AutoCorres.

Word abstraction In Chapter 6 we look at the finite machine-word arithmetic used in the C programming language, and the problems it presents to reasoning. We present a method of automatically and verifiably abstracting signed word arithmetic into arithmetic on unbounded integers, without limiting the end-user's reasoning ability or placing any additional burden on them. We also introduce a method of automatically and verifiably abstracting unsigned arithmetic into arithmetic on unbounded natural numbers; this second abstraction, however, can only be applied if the program being abstracted does not rely on unsigned overflow and the user is willing to prove this.

Additionally, we present some case studies showing how the word abstraction process simplifies reasoning about programs that carry out arithmetic, and also show how the word abstraction process can be extended by the user allowing word-based programming idioms to be abstracted into high-level equivalents.

Heap abstraction In Chapter 7 we describe the difficulties that the byte-level heap representation used by Norrish's C-to-Isabelle parser introduces when trying to reason about programs that access the heap. We go on to describe an existing framework by Tuch et al. [100, 103] that provides mechanisms allowing high-level reasoning on a byte-level heap.

We next implement a simplified version of Tuch et al.'s reasoning framework that is suitable for mechanised reasoning, and then go on to show how we can use this framework to automatically and verifiably rewrite program specifications to operate on an abstract Burstall-Bornat style heap instead of the byte-level heap. Finally, we show how we can soundly combine byte-level reasoning with abstract heap-level reasoning, and present examples of reasoning about our translations.

Evaluation and experience In Chapter 8 we evaluate our algorithms as implemented in the AutoCorres tool. We evaluate AutoCorres in three main ways: (*i*) we show how existing highly abstract proofs about imperative algorithms can be applied to the output of AutoCorres with minimal effort. In particular, we port an abstract proof of the Schorr-Waite algorithm [70] to apply to a concrete C implementation with minimal effort; (*ii*) we next give a qualitative description of how AutoCorres is being used in larger projects, describing the problems it is being used to solve and the challenges that

still remain; finally (*iii*) we quantitatively evaluate AutoCorres by looking at project statistics for some of the larger projects that AutoCorres is being used in.

Conclusion In Chapter 9 we conclude by looking at how AutoCorres helps reduce the burden of reasoning about low-level C, and finally describe some of the problems that still remain to be addressed.

Figure 1.2 depicts the phases carried out by AutoCorres from the initial input C source code to the final output specification, which are described in detail in the following chapters.

2 Related work

Our work looks at how low-level representations of C code can be *automatically* and *verifiably* abstracted into higher level representations that are suitable for *general reasoning*. In this chapter, we look at existing work that attempts to solve similar problems. We focus primarily on existing work that attempts to verify C programs and work that attempts to abstract low-level logical representations of computer programs into higher level logical representations.

2.1 C verification

Formal verification of programs written in the C programming language has been the focus of a great deal of research attention. We can broadly categorise existing tools into three classes: *fully automatic* or *push-button* verification tools, which attempt to prove properties about C programs with little-to-no human interaction; *semi-automatic* verification tools, which require annotations to be added to the source, but will automatically prove them to hold; and *interactive* verification approaches, which allow source code to be reasoned about in an interactive theorem prover, requiring the most effort but also offering the greatest flexibility.

Our categorisation is by no means perfect: some of the fully automatic approaches require annotations or hints to the provers [12]; some of the semi-automatic approaches allow difficult subgoals to be exported to interactive theorem provers [17]; while interactive theorem provers have access to powerful automated solvers. Nevertheless, we feel that our breakdown is broadly in line with the philosophies behind the tools.

2.1.1 Automatic verification of C

The easiest to use class of tools for formal verification of C code are push-button verification tools. The promise is that these tools can be used by developers who have

little or no knowledge about how they work, but still be able to provide a guarantee to the user about their program.

CEGAR The C verification tools Blast [52], MAGIC [28] and SLAM2 [6] use counter-example guided abstraction refinement (CEGAR) [29] to prove various properties about C code. The CEGAR approach begins by constructing a very simple abstract model of the input C program, and attempting to verify properties about it using either an SMT solver or model checker. As counter-examples are found in the overly simplistic model, it is incrementally *refined* until either a legitimate counter-example is found or the desired property has been proven.

While CEGAR can prove quite sophisticated properties of C in theory [7], in practice it is only successfully used at scale in specific domains, such as ensuring that APIs are called in the correct order, avoiding buffer overflows, or avoiding undefined behaviour. Deeper properties such as full functional correctness for larger programs—a problem our work is particularly concerned with—are beyond the reach of such tools at this time.[1] Further, the abstractions generated by CEGAR-based tools are not suitable for generalised reasoning, nor human reasoning. Generalised reasoning is limited because each abstraction is automatically constructed for the purposes of verifying a single property and then thrown away. Human reasoning is limited because CEGAR-generated abstractions are designed to be passed to automated reasoning tools, such as an SMT solvers or model checkers; humans would find the generated abstractions inscrutable. In contrast, the goal of our work is to generate a single abstraction that is not only suitable for *general* reasoning, but also suitable for *human* reasoning.

Shape analysis and separation logic A second class of automatic verification tools use shape-analysis and separation logic [89] to automatically detect data structures used within a C program, infer invariants about these structures, and verify that these inferred invariants are correct. Such tools include include Smallfoot [10], SpaceInvader [40], and Abductor [26]. More recently, Appel's VeriSmall tool [4] has implemented the algorithms used by the Smallfoot tool in the Coq theorem prover, and verified the algorithm against the semantics of the CompCert C compiler [67].

These tools have been successful at a scale, verifying programs with over three million lines of code in a few hours [26]. The properties they are capable of verifying, however, are limited to basic properties about memory safety and other undefined behaviour in C. In the presence of complex logic, control flow, or data-structures, the tools give up analysis on the current function, simply moving onto the next.

Abstract interpretation A third class of tools use abstract interpretation [33] to automatically verify properties about C programs, where the program is analysed by abstracting concrete values into an *abstract domain*. Carefully chosen abstract domains allow certain classes of safety properties to be verified about the program. The ASTRÉE system [35], for instance, is able to verify the absence of undefined behaviour in C code. It has been successfully used to verify this property in large, safety-critical software [12], albeit

[1] The primary limitations are that these deeper properties require either detailed abstractions, which the tools cannot automatically produce in a reasonable number of iterations; or generate abstractions too large for the theorem provers backing the tools to handle.

with program-specific abstract domains being manually specified. The Frama-C framework [98], described in further detail below, similarly has a value analysis phase based on abstract interpretation which is used to discharge simple proof obligations.

Neither ASTRÉE nor Frama-C have verified implementations at this time. Blazy et al. [14] developed a value analysis framework, verified in Coq with respect to the CompCert C semantics [67]; it is able to produce competitive results when compared to the unverified Frama-C framework while providing guarantees of soundness.

While abstract interpretation-based tools have the benefit of being both simple to apply and scalable, this comes at the cost of being limited in the class of properties they are able to verify—like the previous classes of tools described, they remain unsuitable for verifying deeper properties, such as full functional correctness, which our own work is interested in tackling.

The goal of these three classes of C verification tools is quite different from our own: they are capable of verifying large software projects with little user intervention, at the cost of only being able to verify specialised or domain-specific properties about these C programs. This makes the tools practical for large-scale engineering problems faced by industry today, but less useful for *pervasive* verification problems, such as proving full functional correctness. Our own work is concerned with allowing the end-user to prove *any* property about the input C program; this includes deeper properties such as proving full functional correctness, which are well beyond the scope of automated verification tools.

2.1.2 Semi-automatic verification of C

A second category of C verification tools are what we describe as *semi-automated verification tools*. With these, a user typically writes preconditions and postconditions for function entry and exit points, and also annotates loops in the program with invariants and measures. The tools will then analyse the C code and attempt to prove (or disprove) the user's annotations.

VCC One such tool is the VCC tool [31], which parses the user's annotations and attempts to automatically discharge generated proof obligations using the powerful Z3 SMT solver [71] on the backend. The increased automation comes at the cost of reduced expressiveness in annotations, and requires explicit ghost state to guide the reasoner.

VCC's internal implementation has some similarities to our own work. In an attempt to simplify the proof obligations sent to the SMT solver, the VCC tool abstracts C word-based arithmetic and the C heap in ways that are similar to our own abstractions described in Chapter 6 and Chapter 7 [32]. We defer an in-depth discussion of the similarities and differences of the two approaches until Chapter 6 and Chapter 7, but observe that one major point of difference is that the VCC algorithms have only a pen-and-paper proof of correctness, while our own work generates a formal proof of correctness in Isabelle/HOL for every translation.

While our own work's focus is on interactive reasoning, we believe our approach is complementary to that of VCC: one could imagine our tool being used to gener-

ate a verified abstract model of a C program, which automated reasoners could then reason about.

Frama-C The Caduceus framework [43], and its successor the Frama-C framework [98] with the Jessie plugin [72] also supports deductive verification of C. In particular, annotated C code is translated into the functional language named *Why* [44]. The Why platform then generates verification conditions, which the user can choose to discharge in one of a variety of verification tools, including both automated theorem provers (such as CVC3 [9], Simplify [39], and Z3 [71]) and interactive theorem provers (such as Coq [11] and Isabelle/HOL [83]). The framework has been used, for instance, to verify security properties of smartcard implementations [3]. Like VCC, the Frama-C framework's transformations need to be trusted, whereas our own work produces machine-checked proofs of correctness. While the final verification conditions generated by Frama-C can be verified in a theorem prover, the actual process of generating these conditions by Frama-C is unverified. That is, there is no formal guarantee that being able to solve the generated verification condition implies the input program is correct.

More recent work by Herms, Marché and Monate [53], carried out in parallel with our own, resolves this latter problem by developing and formally verifying a verification condition generator (VCG). In particular, Herms et al. developed a formal language similar to that used by Why in the Coq theorem prover. Next, a VCG for this language was developed in Coq and proven to be both sound and complete. Verification engineers can prove programs in this language by first annotating them with preconditions, postconditions and loop invariants; invoking the VCG; and then discharging the resulting verification conditions using an external tool. The most significant difference between Herms et al.'s work and our own is that AutoCorres generates human-readable abstract representations of the input source files, while Herms et al.'s tool only generates verification conditions. While verification engineers reasoning about the output of AutoCorres are able to use our VCG to verify properties about their programs, they also have the option to use more sophisticated techniques, such as proving refinement from a higher level specification, proving non-interference properties, and so on.

2.1.3 Interactive verification of C

A third approach to C verification involves importing C code into a logical representation that is then manually reasoned about by a human using an interactive theorem prover. This approach requires skilled users and greater time investment, but is able to verify deeper properties about the system, such as in the two landmark verifications of the CompCert optimising C compiler [67] and of the seL4 microkernel [57, 82]. The ability to verify deep properties comes from the flexibility provided by the approach: the verification engineer has complete freedom in style and form of properties as well as semantic depth. They can, for instance, reason simultaneously about a program with a VCG [110], prove refinement to a higher level specification [30], or prove more complex properties such as non-interference [74], which are beyond the abilities of existing automatic or semi-automatic C verification tools.

In this section, we look at some of the existing C verification frameworks that focus on interactive verification, as well as describing the larger seL4 verification project which

internally carried out a manual abstraction step in order to achieve the same goals as our own work.

Direct reasoning on C semantics Various other formal semantics of C have been developed, with varying levels of completeness [15, 42, 51, 61, 88]. Two in particular are of interest to us, having both been developed specifically with interactive theorem proving in mind: the CompCert C compiler and the Verisoft C0 compiler.

The CompCert C compiler is a verified C compiler written in the Coq theorem prover [67]. In particular, the proof has a semantics of both the C language and the compiler's target machine code, and has been shown to generate code with observationally-equivalent semantics to that of the input C program [13].

Some early work has taken place using CompCert's C semantics to verify C programs. Dodds and Appel [41], for instance, developed tactics to simplify reasoning about C expressions in CompCert's semantics. Appel also used the Verifiable C semantics—a subset of C connecting to the CompCert backend—to verify an implementation of the SHA-256 cryptographic hash function; the proof made heavy use of symbolic execution of the Verifiable C semantics, and internally used separation logic to specify function behaviours [5]. Both of these proofs reason directly about the low-level semantics generated by their respective parsers, using tools to automate some of the more tedious proof obligations. Our aim is to abstract the low-level C semantics, eliminating the need for such tools.

The Verisoft project [2] aimed to carry out a 'full stack' proof of correctness, verifying a processor, compiler, and operating system, and linking the proofs together to form an end-to-end guarantee of correctness. One aspect of the project was a C-like compiler, which compiled a subset of the C programming language named C0 [65, 66]. Various programs were verified in Isabelle/HOL using the C0 semantics, including string libraries, linked list libraries and a simple text-based email client. The proofs of correctness took place directly on the C0 semantics, using a Hoare-logic specification language and verification condition generators to generate proof obligations. These were, when possible, discharged using external model checking tools [37]. Like the CompCert framework, the Verisoft project reasons directly on their low-level C semantics instead of a more abstract representation of it as is done in our work.

The seL4 proof methodology The seL4 project used the Isabelle/HOL theorem prover to show that the 10,000 line C implementation of a small operating system kernel was a refinement of a high-level abstract specification [57, 82]. This refinement theorem was originally used to prove that the C implementation of seL4 only exhibited behaviours that were present in its formal specification. Subsequent work used the proof to show further properties about the kernel's C implementation—such as integrity and non-interference—simply by reasoning about the abstract specification [74, 96].

The C implementation of seL4 was imported into Isabelle/HOL using the same C-to-Isabelle parser used in our own work, which we discuss further in Section 3.4. The abstract specification of seL4 was hand-written directly in Isabelle/HOL. The proof of refinement between the output of the C-to-Isabelle parser and the abstract specification took place in two phases, using a hand-written intermediate specification named the *executable specification* as a stepping stone [30, 110].

The proof of refinement between the C implementation and the executable specifica-

tion is the most closely related to our own work. Winwood et al. described the details of this step in [110]. While some basic tools were developed to automate the refinement between the output of Norrish's C-to-Isabelle parser and the executable specification, for the most part the proof was a manual effort, requiring approximately 3 person-years of effort, excluding the time required to build logical frameworks and tools [57].

Our own work was initially undertaken as a method to automate the low-level seL4 refinement proof, and hence has the same goal as this part of the proof; that is, abstracting low-level C semantics into a high-level logical representation. Unlike the manual proof in seL4, however, our work automates both the *creation* of the executable specification and the *proof of refinement* between the output of the C-to-Isabelle parser and the generated specification.

2.2 Abstraction of low-level semantics

In this section, we look at work that has investigated the problem of taking low-level representations of programming languages and abstracting them into higher level representations for the purposes of further reasoning. Two areas in particular that relate closely to our own work are verified assembly decompilation and Yin et al.'s *Echo Framework* [111], which we describe below.

Assembly decompilation Myreen et al. [76–78] investigated the problem of abstracting the low-level semantics of real-world ARM machine code programs into functional representations in the HOL4 theorem prover. Individual instructions are decoded into logic and chained with surrounding instructions to form functional blocks. Assembly instructions forming loops are converted into tail-recursive functions in HOL4, which do not require termination proofs.

Similarly, Li [68] uses the HOL4 theorem prover to decompile of a small subset of the ARM instruction set, in the context of compilation verification. Like our own work, Li used a monadic representation for the intermediate representation of the programs, using tail-recursive functions of HOL4 to model loops.

Both Myreen et al. and Li's work share commonalities with our own translations from low-level semantics into shallow embeddings. At a high-level, our work concerns not only in the initial conversion from low-level semantics into logic, but also further abstractions such as the heap abstraction and word abstraction phases described in Chapter 6 and Chapter 7, respectively. We further discuss the differences in Chapter 4.

Reverse synthesis Yin et al. [111] developed a technique termed *reverse synthesis* to carry out high-level reasoning about SPARK Ada programs in the PVS theorem prover. Their technique involved (*i*) annotating the program implementation with preconditions, postconditions and loop invariants; (*ii*) verifying that the implementation satisfies the annotations; (*iii*) mechanically extracting the annotations to form a specification in an interactive theorem prover; and finally (*iv*) using the theorem prover to show that the extracted specification satisfies the properties of a high-level specification of the program. To simplify the annotation proofs, Yin et al. additionally carried out semantics-preserving

transformations on their input program, such as reversing loop unrolling optimisations or replacing iteration with recursion.

While the goals of our work are similar to that of Yin et al.'s, our work differs by removing the need for hand-written annotations before the program can be imported into a theorem prover. Our work also has an end-to-end proof, while Yin et al.'s framework carried out program transformations and specification extraction using programs that needed to be explicitly trusted, while each step in our work generates a proof of correctness in Isabelle/HOL.

Conversions between language and logic We are aware of two other projects that investigate how deeply embedded representations of programming languages can be verifiably translated into shallowly embedded logical representations, or *vice versa*.

The *CakeML* project [62] is a verified compiler for a subset of the Standard ML programming language, translating ML programs into x86-64 machine code. One aspect of the project developed tools allowing shallowly embedded HOL terms to be verifiably translated into a deeply embedded representation of ML code [79], which in turn can then be compiled into executable machine code; in effect, this allows programs to be written in HOL, compiled to machine code, and then executed in a trustworthy manner.

Similar work carried out in parallel to our own work by Myreen [75] similarly showed how deeply embedded representations of Lisp programs could be verifiably translated into shallowly embedded HOL representations, and *vice versa*.

Both of these works which convert between deeply embedded and shallowly embedded representations of programming languages have similarities to our own work described in Chapter 4. The most significant difference between these works and our own is that we focus on converting imperative languages, which raises difficulties not present in functional programs; these include the representation of local variables, the representation of the heap, and the translation of imperative-style loops. Further, our own work carries out further abstractions that are more specific to the C programming language, such as heap abstraction and word abstraction, described in Chapter 7 and Chapter 6, respectively.

In summary, existing C verification frameworks typically focus on either providing a convenient translation of C into logic or a conservative, trustworthy translation. Our goal is to combine the two.

In the next chapter, we present the existing work our own work builds upon, including a description of the C programming language, the Isabelle/HOL theorem prover, and the conservative C-to-Isabelle parser we use in our work.

Chapter Summary

- Existing C verification frameworks typically focus on either providing a *convenient* translation of C into logic, or a *conservative* translation. Convenient frameworks automatically abstract low-level languages into higher level representations without providing guarantees of correctness. Conservative frameworks provide high-assurance reasoning directly on low-level program representations. Our work aims to combine the two.
- Existing work on automatic abstraction of program semantics exists in the context of assembly code verification [68, 78]. While there are similarities to our own work, particularly in our initial transformations, our broader aim is to carry out higher level abstractions such as simplifying control flow, abstracting word arithmetic, and abstracting the model of the heap. This is not a concern of current assembly abstraction frameworks.
- Abstraction of program semantics has also taken place in the context of SPARK Ada verification [111]. While promising, this work requires every function to be manually annotated before it can be applied. Additionally, the produced output is not ideal for general human reasoning. Our work aims to automatically provide an output suitable for general, human reasoning, while also providing an end-to-end proof of its correctness.

3 Background

The goal of this thesis is to simplify reasoning about C code by automatically abstracting low-level logical representations of C into a higher-level representation. In this chapter, we lay the foundations of our work by introducing the existing research being built upon. In particular, we describe the following four works:

- **The C programming language:** Section 3.1 gives a description of the *C programming language*, along with some of its quirks, such as undefined- and implementation-defined behaviour;
- **Isabelle/HOL:** Section 3.2 gives an introduction to the Isabelle/HOL interactive theorem prover, including notation used in the rest of this document;
- **The Simpl language:** Section 3.3 gives a description of Schirmer's Simpl language, which allows imperative programs to be modelled in Isabelle/HOL, and is the input language to our own work; finally
- **C-to-Isabelle parser:** Section 3.4 introduces Norrish's C-to-Isabelle parser, which conservatively translates C into Norrish's Simpl language enabling formal reasoning about C.

Together, these four works form the conservative, low-level input to our own work described in the following chapters.

3.1 The C programming language

It may seem a little strange that in 2014 we are still talking about the C programming language [55], an imperative general-purpose programming language initially developed in the late 1960s.

When the language was originally developed by Dennis Ritchie between 1969 and 1973 [90], C was distinctive for several reasons: (*i*) it had a (for the time) strong static type system; (*ii*) the language was relatively *portable*—that is, code written for a particular system could be recompiled and used on a different type of system with relative ease; and (*iii*) the operations provided by the language were grounded in those provided by real machines, allowing the language to be compiled into efficient code. It is clear why these advantages helped the C language to boom in popularity in the 1970s and 1980s, becoming the language of choice in both application and systems programming.

In 2014, however, it would seem that C has been thoroughly superseded by newer languages such as C++, Java, C#, and so on. These newer languages, despite having obvious roots in C, add significant programmer conveniences such as object-oriented programming primitives, stronger type systems (which are able to provide certain correctness guarantees, unlike the type system of C), increased portability, automatic memory management, and so on.

Despite the C language's decreased popularity as a general-purpose application programming language, C has remained the language of choice in areas such as embedded systems, systems-level programming, real-time programming and safety-critical programming. One reason for this is simply due to existing legacy code: because a large amount of code already exists written in C, it is easiest to continue using C. The C programming language also has benefits for certain new projects; such benefits arise from a combination of important properties:

Small language size The C language's syntax is relatively small, making it easier to develop tools for the language.[1] For instance, C has multiple formal models of the language [15, 42, 51, 84, 88], verified compilers [67], model checking tools [6, 52], and formal analysis tools [85, 98]. In contrast, the C++ programming language, which is both larger in scope and more difficult to parse [109, p. 147], has far fewer tools despite its similar popularity to C. While some formal models of the language exist [106], they only cover a small subset of the language.

Small language runtime The C language has a small language runtime, which is particularly important in safety- and security-critical contexts. Language runtimes form part of the trusted-computing base; that is, every program written in a particular language implicitly needs to trust the runtime for its correctness. In the context of safety- and security-critical programs, a smaller language runtime means less code that needs to be verified, audited and/or implicitly trusted.

In C, the only strictly necessary runtime requirement of a minimal C program is a functioning stack. The majority of the C standard library, such as `memcpy` and `memset`, can not only be written in C itself but will typically only need to be trusted if used.[2]

[1] For instance, it was common for obscure embedded platforms to ship with a C compiler but not a C++ compiler, because the former was far easier to develop than the latter. With the development of several open source multi-architecture C++ compilers however, it has become easier for vendors to ship both a C and C++ compiler in recent years.

[2] A notable exception to this is that some C compilers will emits calls to library functions as part of their compilation process. For instance, `gcc` will emit calls to `memcpy` and `memset` instead of emitting code to copy or zero large structures. Such internally emitted calls can typically be disabled during compilation if explicitly requested.

In contrast, even simple implementations of languages such as Java and C# require garbage collectors (typically consisting of thousands of lines of code), support for stack unwinding (for exceptions), object introspection libraries (for language reflection) and a substantial amount of standard library code.

Predictability Compiled C programs are typically more predictable in their memory usage and timing characteristics than programs developed in more modern languages. This predictability of C is primarily due to its lack of automatic memory management.

Code with predictable timing characteristics is particularly important in real-time systems, where a calculation that arrives too late is often as bad as an incorrect calculation; while code with predictable memory usage is required in safety-critical systems to ensure that the system will not run out of memory at an inopportune moment.

Modern languages such as Java and C# typically carry out implicit memory allocations, which may lead to seemingly innocent statements triggering out-of-memory exceptions. They also implement automatic memory management using garbage collectors, which may introduce undesirable latencies during time-critical parts of a program.

By forcing the programmer to explicitly deal with all memory allocation and reallocation issues, C remains a more predictable language.

Efficiency C types and operations are well-grounded in those provided by real machines. The standard types in C such as `int`, `unsigned long` and pointers map directly to CPU registers, while operations on these types typically map to single CPU instructions. These efficiencies are important in domains where performance is critical, such as in operating system development or language runtime implementations.

In some sense, it is the spartan nature of the C language that makes it suitable for real-time and safety-critical domains: by offering few programmer conveniences or abstractions over the hardware, it is possible for programmers to know precisely what is going on behind the scenes.

3.1.1 Features of C

For the most part, the C programming language is relatively conventional: it has standard control-flow primitives, such as `if`/`else` conditionals, `while` and `for` loops, the `goto` statement, and so on. We will not attempt to provide a full description of the C language, but instead refer interested readers to Kernighan and Ritchie's guide of the language [56]. Briefly, some of the features of the language include:

- Functions, along with support for recursion and mutual recursion. Higher-order functions may be crudely emulated using function pointers;
- A statically checked but weakly enforced type system, where the programmer may freely override the type-system using *type-casts*;
- Pointers, which allow references to objects in the language. Pointers are typically implemented by storing the memory address of the object they reference. C supports arithmetic on pointers (such as addition, subtraction, and comparison); has a `NULL` pointer, used to indicate the absence of an object; and supports casting pointers to and from integers, which in turn allows low-level access to memory;

```c
int sum(int n) {
    int result = 0;
    while (n > 0) {
        result += n;
        n--;
    }
    return result;
}
```

(a) Sum of numbers

```c
#include <stdio.h>

int main(void)
{
    printf("Hello, world!\n");
    return 0;
}
```

(b) Hello world

```c
struct node {
    struct node *next;
    long data;
};

long add_list(struct node *head) {
    long r = 0;
    while (head != NULL) {
        r += head->data;
        head = head->next;
    }
    return r;
}
```

(c) Linked list

Figure 3.1: Three simple C programs. `sum` returns the sum of numbers from 1 to n; `main` writes the string `Hello, world!` to output; while `add_list` adds the values in a linked list.

- Manual memory management, where heap memory is allocated with a library function `malloc` and later released by an explicit call to the library function `free`. Memory required for local variables and statically declared variables are automatically managed by the compiler;
- Finite-length signed and unsigned integer types, typically implemented with machine words for efficiency;
- Finite-precision floating point types, typically implemented with machine words if available, or emulated by libraries included with the C compiler;
- Compound/aggregate types, such as arrays, `structs` (containing multiple named types), and `unions` (containing multiple named types, of which only one is valid at any time);
- Support for multi-threading, including atomic types, thread-local storage and memory ordering operations; and
- A standard library, supporting basic I/O, string manipulation, memory management, etc.

Three simple programs that show some of these language features are shown in Figure 3.1. The first is a function that sums the numbers from 1 to *n*; the second prints the string `Hello, world!` to output; while the third iterates over a linked list, summing the values stored in the nodes.

Notable language features that are *not* part of the C language include automatic memory management (such as garbage collection or reference counting); array bounds checking; a native string type (which is instead implemented using arrays of bytes); object-oriented abstractions, such as classes or inheritance; and exceptional control flow.

While our work supports reasoning about a large subset of the C programming language, we do not support the entire specification; Section 3.4 describes in detail that subset of C that our work supports.

3.1.2 Undefined and implementation-defined behaviour

The C standard specifies that certain operations that can be performed by a C program will result in *undefined behaviour*, where the compiler is free to exhibit any behaviour it chooses. Examples of actions which lead to undefined behaviour include:

- Division by zero (C11, 6.5.5/5);
- Out-of-bounds array accesses (C11, 6.5.3.2/4);
- Executing a signed arithmetic operation that either overflows or underflows (C11, 6.5/5);
- Expressions that attempt to modify the same variable more than once, such as `(i = 1) + (i = 2)` (C11, 6.5/2);
- Program execution reaching the end of a non-`void` function without a value being returned, and then the return value being subsequently used (C11, 6.9.1/12);
- Reading from an uninitialised local variable (C11, 6.3.2.1/2); and

- Attempting to access a pointer to an object which is now out of scope (C11, 6.2.4/2).

While the phrase "the compiler is free to exhibit any behaviour it chooses" perhaps sounds a little vague, the C standard unfortunately doesn't commit the C compiler to anything more. In practice, typical behaviours are the program crashing, or silently producing incorrect results;[3] the compiler is well within its rights to set the user's keyboard on fire, however. It is the programmer's responsibility to ensure that actions with undefined behaviour do not occur.

The C language also defines certain operations which have *implementation-defined* behaviour. These operations may differ between compilers and platforms, but will be precisely defined for each particular compiler running on a particular platform.[4] Examples of implementation-defined behaviours include:

- The size and range of types such as `char`, `int` and `long` (C11, 5.2.4.2.1);
- Whether a right-shift operation on a negative value such as `(-1 >> 1)` will sign-extend or not (C11, 6.5.7);
- If the `char` type is signed or unsigned (C11, 6.2.5);
- What happens when a large unsigned integer is cast to a smaller signed integer (such as `(signed short)65537`) (C11, 6.3.1.3);
- The result of casting a pointer to an integer, or vice versa (C11, 6.3.2.3).

In our work, the implementation-defined details of C are modelled to match the GNU C Compiler [99] for the 32-bit ARMv6 architecture.[5] Our work for most the part is independent of these assumptions, though some assumptions—such as a `char` having 8-bits and signed integers having a two's complement representation—are baked rather deeply into our tool's concrete implementation.

3.2 The Isabelle/HOL interactive theorem prover

Formal program verification typically consists of long, tedious proofs. Not only are these proofs time consuming to construct, but they also face the problem that even a trivial mistake may lead to an incorrect conclusion. For this reason, modern program verification projects tend to use *theorem provers*, which can both assist humans to produce proofs and verify that such proofs are correct.

[3] Modern compilers will assume that actions leading to undefined behaviour cannot occur, and use these assumptions to optimise code. This often leads to faster code, but also causes subtle and seemingly innocuous bugs that lead to very surprising behaviour [105].

[4] Such implementation-defined behaviours will ideally—but, unfortunately, not necessarily—be specified in the compiler's documentation.

[5] In particular, the `char`, `short`, `int` and `long` types are defined to be 8-bit, 16-bit, 32-bit and 32-bits, respectively; the `char` type is considered to be `unsigned` unless explicitly marked `signed`; and all integer types are represented as two's complement.

In this work, we use the *Isabelle/HOL* [83] theorem prover. Isabelle is a generic interactive theorem prover that supports multiple logics; Isabelle/HOL is the version of Isabelle instantiated to use higher-order logic (HOL).

Isabelle uses an LCF-style *proof kernel* [46, 47] to ensure correctness. In particular, Isabelle has a small trusted core which is responsible for keeping track of proven facts and determining if new logical inferences are valid. More complicated procedures can be written to simplify or automate proof generation, but they can only generate new facts by appealing to the primitive interface of the proof kernel. This kernel-based approach allows users to extend Isabelle without needing to worry that defects in such extensions will lead to unsoundness.[6] While the proof kernel must still be trusted, the amount of faith that must be placed in it is minimised because it is both small and conservatively constructed.

3.2.1 Interacting with Isabelle

Users typically interact with Isabelle using the Isabelle/jEdit user interface [107], pictured in Figure 3.2. The top panel consists of a text editor where proof commands are written by the user. The bottom panel displays the state of Isabelle at the location of the cursor, which is typically where the user is actively writing their current proof.

A typical Isabelle/HOL proof begins with the keyword `lemma`, followed by the predicate that the user wishes to prove. There are two main styles of proof in Isabelle: *apply-style proofs*,[7] which are a backward-reasoning style of proof; and *Isar proofs* [108], which are a forward-reasoning style of proof with a focus on readability. Isar proofs have the advantage of being somewhat human-readable, at the cost of being more verbose than their apply-style equivalents. An example of each style is given in Figure 3.3.

Isabelle, written in the Standard ML programming language, also offers an ML API for interfacing with its proof kernel. This means that users can write their own tactics and extensions by interfacing with the proof kernel. The API allows users to programmatically create new definitions, state theorems and proceed to prove them, analyse existing proof statements, and so on.

The AutoCorres tool described in this paper uses both an extensive library of hand-written proofs and also a large amount of ML that interacts with Isabelle directly. Hand-written definitions and theorems provide a foundation that is then used by AutoCorres' ML code-base to automatically carry out larger proofs.

3.2.2 Isabelle's meta-logic

As mentioned earlier, Isabelle is an interactive theorem prover designed to support multiple types of logic, of which HOL is just one. To support several such logics, Isabelle implements a *meta-logic*, which describes what logical inferences are valid. For example, in English, we might write the statement

[6] This doesn't, of course, prevent the user from proving *useless* theorems—but any such useless theorem will at least be *true*.

[7] The name *apply-style* comes from the keyword `apply` that is used at the beginning of each proof step.

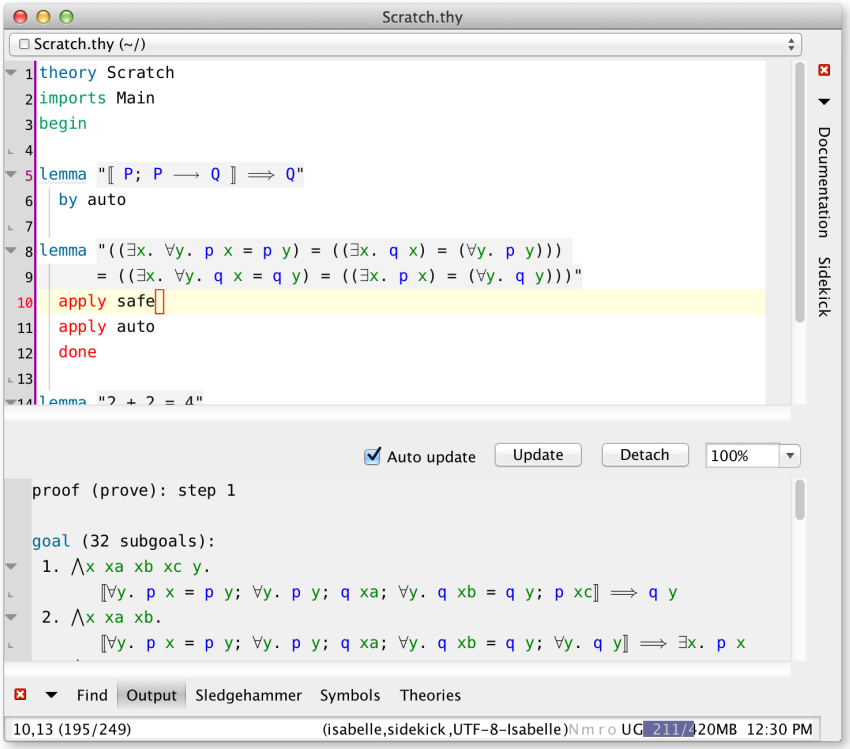


Figure 3.2: A screenshot of the Isabelle/jEdit IDE [107], version 2013-2 running on MacOS X. Proof commands are written in the top panel, while the output of the Isabelle theorem prover is shown in the bottom panel. The output currently shown is the state of Isabelle at the point of the cursor, highlighted yellow in the image.

$$\textit{If } P \longrightarrow Q \textit{ is true, and } P \textit{ is also true, then } Q \textit{ is true.}$$

Here, the English words "if", "then", and "is true" are meta-logical, while the mathematical terms P, Q, and $P \longrightarrow Q$ are logical. In Isabelle, this statement would instead be written as

$$\llbracket P \longrightarrow Q;\ P \rrbracket \Longrightarrow Q$$

Here, the arrow "$\Longrightarrow$" is a *meta-implication* operator that separates the premises of the statement from its conclusion. The brackets "$\llbracket$" and "$\rrbracket$" simply group the assumptions on the left-hand side of the meta-implication operator.

Isabelle additionally implements a *meta-universal quantifier* (or the *meta-forall*) operator, which states a theorem is true for any value of the named variable. For example, the following rule describes mathematical induction over the natural numbers:

$$\llbracket P\ 0;\ \bigwedge n.\ P\ n \Longrightarrow P\ (n + 1) \rrbracket \Longrightarrow P\ n$$

That is, if a property P is true for the natural 0; and if the property P being true for n implies that it is also true for $(n + 1)$; then the property is true for all values of n.

Here, the symbol "$\bigwedge$" is the meta-forall quantifier, stating that the assumption must hold for all values of n.

```
lemma "¬ prime (6::nat)"
  apply (clarsimp simp: prime_nat_def)
  apply (rule exI [where x=3])
  apply clarsimp
  done
```

(a) Apply-style proof

```
lemma "¬ prime (6::nat)"
proof (clarsimp simp: prime_nat_def)
  obtain x :: nat where "x = 3" and "x dvd 6"
    by auto

  thus "∃m :: nat. m dvd 6 ∧ m ≠ Suc 0 ∧ m ≠ 6"
    by fastforce
qed
```

(b) Isar proof

Figure 3.3: Two Isabelle/HOL proofs that the number 6 is not prime, using (a) apply-style reasoning and (b) Isar reasoning.

In this document, we typically use the more standard notation

$$\frac{P\ 0 \qquad \forall n.\ P\ n \longrightarrow P\ (n+1)}{P\ x}$$

to describe such statements. The main exception is when we are displaying literal output from Isabelle, in which case we will use Isabelle's default meta-logic notation.

3.2.3 Notation

Isabelle/HOL mostly uses standard mathematical notation. In this section, we describe notation used in Isabelle/HOL that departs from such standards, or notation that is simply less commonly known.

Functions and types The type of a term is written using the notation $a :: T$, which states that term a has type T. *Type variables*, which may be instantiated to concrete types, are given the notation $'t$. In Isabelle/HOL, every type has at least one element in it; users defining new types must prove that they are non-empty before Isabelle will accept the definition. The constant named undefined is defined for every type, and represents a arbitrary (but fixed) element of the type.

Functions in Isabelle/HOL have types of the form $f :: 'a \Rightarrow 'b \Rightarrow 'c$, stating that function f accepts two parameters having type $'a$ and $'b$ respectively, and returns a value of type $'c$. The function id is the *identify function* with definition $(\lambda x.\ x)$. Finally, the notation $f(a := b)$ is used to represent function update, with the definition

$$f(a := b) \equiv \lambda x.\ \text{if } x = a \text{ then } b \text{ else } f\ x$$

Finally, Isabelle/HOL has a type $'a\ itself$ which can be generated from expressions of the form $\mathsf{TYPE}('a)$ for every type $'a$. This allows users of Isabelle/HOL to write functions that accept a type as a parameter; we will see this used in Chapter 7.

Sets For most the part, Isabelle/HOL uses standard mathematical set notation. $\emptyset$ represents the empty set; $\{n.\ n < 10\}$ represents the set of naturals less than 10; while UNIV represents the universal set, containing all elements of the associated type. Sets have types of the form $'a\ set$, where $'a$ is the type of the elements in the set.

Lists Lists in Isabelle/HOL are represented using the notation $[1, 2, 3, \ldots]$. The empty list is $[]$; elements are added to the front of the list using $x{\cdot}xs$; and lists are appended using $xs\ @\ ys$. The function $\mathsf{hd} :: 'a\ list \Rightarrow 'a$ returns the first element of a list, while $\mathsf{tl} :: 'a\ list \Rightarrow 'a\ list$ returns the remainder; $\mathsf{hd}\ []$ returns undefined.

The option type Isabelle/HOL has an *option type*, denoted $'a\ option$, that can either have the value Some x or the value None, where x has type $'a$. The function $\mathsf{the} :: 'a\ option \Rightarrow 'a$ is defined such that $\mathsf{the}\ (\mathsf{Some}\ x) = x$. The option type is useful for data that may not be present, and is also used to represent partial functions in Isabelle/HOL. For example, the function $f :: 'a \Rightarrow 'b\ option$ is a function that takes an argument of type $'a$ and optionally returns a value of type $'b$.

The unit type Isabelle has a *unit* type, consisting of only a single element $()$, known as the *unity*. The type has the property $\forall v.\ v = ()$.

Tuples Tuples combine two or more elements into one type, and use the standard mathematical syntax $(a, b) :: 'a \times 'b$. The functions fst and snd access the first and second elements in a tuple respectively. In particular:

$$\mathsf{fst}\ (a, b) = a$$
$$\mathsf{snd}\ (a, b) = b$$

Records Isabelle/HOL supports records, which are analogous to tuples where each element is given a name. A record with fields x, y, and z is constructed with the notation $(\!|\, \mathsf{x} = 1, \mathsf{y} = 2, \mathsf{z} = 3 \,|\!)$. Field x of record r may be accessed with $\mathsf{x}\ r$, and updated with the notation $r(\!|\, \mathsf{x} := 1 \,|\!)$. Each field of a record $'r$ additionally has an update function with the name of the form F_update and type $('a \Rightarrow 'a) \Rightarrow 'r \Rightarrow 'r$, where the first argument is a function that takes the old value of the field and returns a new value.

Other notation used in this document will be described as it appears.

3.3 Simpl: Modelling imperative programs in Isabelle/HOL

Before we can formally reason about imperative programs in Isabelle/HOL, we must first represent them in our logic. Whatever representation we use needs to be able to correctly model various behaviours commonly seen in imperative programs such as:

- Loops, including nested loops;
- Function calls, including recursive and mutually recursive function calls;
- Reading and writing objects in memory;

- Faults, such as when programs divide by zero or access invalid memory; and
- Non-determinism, frequently required to model program interactions with hardware and the outside world.

Schirmer's *Simpl* language [92, 93] is a small, generic language that allows imperative programs using such features to be modelled in Isabelle/HOL. The language has a large library of theorems proven about it, and also comes with a number of tools, such as a verification condition generator, that assist with reasoning about Simpl programs. The language was designed with the aim of being able to model programs written in a variety of languages, such as C, Java or Ada. In particular, the language allows standard imperative language primitives such as function calls, while loops, and memory to be modelled in Isabelle/HOL.

Simpl uses deeply embedded statements and shallowly embedded expressions; that is, statements (such as `while` loops and `if` statements) are encoded structurally, while expressions (such as $x + 2 > 4$) are encoded directly using Isabelle expressions and types. Simpl's deeply embedded statements simplifies modelling and reasoning about non-terminating and recursive programs,[8] while also allowing meta-reasoning about the Simpl language itself. The use of shallowly embedded expressions allows the user flexibility in determining how variables and types should be modelled.

Simpl program and execution states

Simpl programs have two important states that are tracked during execution: a *program state*, which represents the memory of the program; and an *execution state*, which represents what mode of execution the program is in.

Simpl does not dictate the type of the program state, instead leaving the choice to the user. Simpl's execution states, however, are fixed, and defined as follows:

$$\begin{aligned}&\textbf{datatype}\ ('s,\ 'f)\ \mathit{xstate} = \\ &\quad \mathsf{Normal}\ 's \\ &\quad \mid \mathsf{Abrupt}\ 's \\ &\quad \mid \mathsf{Fault}\ 'f \\ &\quad \mid \mathsf{Stuck}\end{aligned}$$

The two standard execution states of Simpl are Normal *s*, which indicates that the program is executing in normal imperative mode; and Abrupt *s*, which indicates that the program is currently propagating an exception and will thus not execute any instructions until the exception is caught. Both the Normal and Abrupt execution states have a parameter *s* which represents the current program state.

Simpl also includes two error execution states: Fault *f* indicates that the program has irrecoverably failed with a failure code *f*. Stuck indicates that program execution has reached a point where there are no successor instructions to execute; this might happen, for example, when the program attempts to call a function that does not exist.

[8] In particular, every Isabelle function must have a proof of termination before its definition will be admitted by Isabelle; deeply embedded representations of functions can side-step this issue. We return to this problem of termination in Section 4.3 when we attempt to convert the deeply embedded Simpl programs into shallowly embedded monadic representations.

Simpl statements

The Simpl language consists of 11 different commands, representing various language constructs such as loops, conditional branches, and updates to the program's state. Each command has the type

$$('s, 'p, 'f)\ com$$

where $'s$ is the type of the state, $'p$ is the type used to represent the names of functions, and $'f$ is the type of faults.

The statements in the Simpl language are as follows:

Basic m — Update the program's state from s to $m\ s$. This is used to write to a program variable or modify the program's heap, for instance.

In the common case where m is a function that updates a local variable `a` to the value b, we use the notation 'a :== b.

Skip — This command is a 'no-op', making no changes to the program state. It is typically used as a placeholder in locations where a statement is syntactically necessary but not desired, such as in the `else` clause of an `if` statement. It is semantically equivalent to Basic $(\lambda x.\ x)$.

Guard $z\ C\ c$ — Assert that the current program state s is in the set C, and then execute c. If $s \notin C$, the program enters the execution state Fault z, representing irrecoverable failure.

Guards are typically used to ensure that the program satisfies certain correctness conditions. For example, before dereferencing a pointer `p`, we may wish to assert that `p` is non-`NULL`. If the pointer is `NULL`, then the program is considered to have failed. The fault parameter z is used to indicate the reason for the failure; for instance, a guard for a divide-by-zero error may use a different fault parameter to a guard for a `NULL`-pointer dereference to allow the different fault types to be reasoned about individually.

Spec r — Non-deterministically select a new state s' based on the current state s such that $(s, s') \in r$. If there is no state s' satisfying the relation, the program enters the Stuck execution state. Such non-determinism is used to model interactions with hardware, uninitialised memory, or to allow execution of under-specified procedures.

c_1;; c_2 — Execute c_1 followed by c_2. The second half will not be executed if the program state is Abrupt, Fault or Stuck.

Cond $e\ c_1\ c_2$ — An *if-then-else* construct. Determine if the current program state $s \in e$. If so, execute c_1; otherwise, execute c_2.

While $e\ c$ — A simple while loop. While ever the current program state $s \in e$, execute the loop's body c. If the loop never terminates, the While statement will have no successor states.

Throw — Update the program's execution state from Normal s to Abrupt s. The latter represents *abrupt execution*, and is used to model exceptions, `break`

Table 3.1: Concrete syntax used for Simpl programs

Simpl Command	Syntax
Basic m	**BASIC** m
Guard z C c	**GUARD** z C c
Spec r	**SPEC** r
Cond e c_1 c_2	**IF** e **THEN** c_1 **ELSE** c_2 **FI**
While e c	**WHILE** e **DO** c **OD**
Throw	**THROW**
Catch c_1 c_2	**TRY** c_1 **CATCH** c_2 **END**
Call p	**CALL** p
DynCom c_s	**DYNCOM** c_s

and `continue` statements in loops, `return` statements in the middle of a function, etc.

Catch c_1 c_2 — An exception-handling construct. Execute c_1. If the resulting program mode is Abrupt s, the program mode is changed to Normal s and c_2 is executed. Otherwise, c_2 is simply skipped.

Call p — Call the procedure labelled p. When executing Simpl programs, an *execution context* $\Gamma :: 'p \Rightarrow ('s, 'p, 'f)$ *com option* is provided which maps procedure labels to their bodies. When a Call p command is executed, $\Gamma\ p$ is evaluated. If $\Gamma\ p =$ Some b, then the execution continues by evaluating b. If $\Gamma\ p =$ None, however, the program enters the Stuck execution state.

DynCom c_s — Execute dynamically generated Simpl code. In particular, the parameter $c_s :: 's \Rightarrow ('s, 'p, 'f)$ *com* is a function that takes the current program state and returns a program body. When a DynCom command is executed, the current program state is passed to the function c_s and then the resulting body is executed.

DynCom was designed by Schirmer to allow programs that dynamically load and execute code to be modelled in the Simpl language. In this work, however, our use of DynCom is far more modest, using it simply as a building block for modelling C function calls.

When listing larger Simpl blocks, we use the additional syntax for control flow statements, such as

IF e **THEN** c_1 **ELSE** c_2 **FI**

for Cond e c_1 c_2, and so on. The full table of syntax is shown in Table 3.1.

Reasoning about Simpl programs

When reasoning about Simpl in this document, we use Schirmer's formal big-step semantics. The notation

$$\Gamma \vdash \langle C, s \rangle \Rightarrow t$$

is used to state that the program C, starting from execution state s has at least one path reaching the execution state t. The variable Γ maps function names to function bodies, and is used for making function calls in Simpl. The formal big-step semantics of each of the Simpl statements described above are provided in Appendix A.1.

We additionally use Schirmer's notation

$$\Gamma \vdash C \downarrow s$$

to specify that all paths of the program C starting in execution state s will eventually terminate. The formal definition of termination of Simpl programs is provided in Appendix A.2.

3.4 Translating C into Isabelle/HOL

While Schirmer's Simpl language provides a way to allow imperative programs to be modelled in Isabelle/HOL, if we want to reason about concrete C programs we still need to somehow translate them into Simpl.

Our work uses Norrish's C-to-Isabelle parser [85, 103] which parses C code and conservatively translates it into the deeply embedded Simpl language described above. Norrish's work attempts to carry out a *conservative translation* from C to Simpl; that is, it attempts to precisely model C at the cost of generating an output that will potentially be harder to reason about.

Because the C-to-Isabelle parser generates a conservative representation in Isabelle/HOL, recent work by Sewell and Myreen [95] has successfully used SMT solvers to show that the binary output of the `gcc` compiler matches the model generated by the C-to-Isabelle parser for a number of large programs. Such work could be used to allow us to reason about the output of Norrish's C-to-Isabelle parser and compile our programs using `gcc` without needing to trust either tool—the final proof links the formal Isabelle model of C to the generated assembly.

Norrish's parser supports a large subset of C, including:

- Loops, including `for`, `while` and `do` loops, `break` and `continue`;
- Well-formed `switch` statements that do not use fall-through (i.e., each non-empty `case` in a `switch` statement must end with either a `break` or `return` statement);
- Function calls, including recursive and mutually recursive calls;
- Word types and word arithmetic, which are accurately modelled;
- Some aggregate types, including structures and arrays, but excluding unions and bitfields; and
- Pointers, including pointer arithmetic and casts between pointer types.

The C-to-Isabelle parser does not support the full C standard, with notable exclusions including:

- `goto` statements;

```
int max(int a, int b) {
    if (a <= b)
        return b;
    return a;
}
```

```
max_body ≡
    TRY
        IF {| 'a ≤s 'b |} THEN
            'ret__int :== 'b;;
            'global_exn_var :== Return;;
            THROW
        ELSE
            SKIP
        FI;;
        'ret__int :== 'a;;
        'global_exn_var :== Return;;
        THROW;;
        GUARD DontReach ∅
            SKIP
    CATCH
        SKIP
    END
```

Figure 3.4: A simple `max` function and its translation into Isabelle/HOL's Simpl language by Norrish's C-to-Isabelle parser.

- Expressions with side-effects;
- References to local variables;
- Concurrency;
- Signals and signal handling;
- Floating point types or arithmetic; and
- Very limited support for function pointers, only supporting functions that have no arguments and a `void` return type.

Despite these limitations, the C-to-Isabelle parser is still very useful, having been used to formally verify a full operating system microkernel [57], non-trivial graph algorithms [86], a real-time operating system [80], as well as many other projects described further in Section 8.2.

The C-to-Isabelle parser models word arithmetic to match a two's-complement 32-bit system. The memory model used by the C-to-Isabelle parser is configurable, though our work exclusively uses Tuch's instantiation [102, 103], which models memory as a function from 32-bit addresses to 8-bit bytes, and explicitly defines how each object in the system is encoded and decoded to and from bytes. Full details are provided in Chapter 7.

As mentioned earlier, in our work the implementation-defined details of C, such as `struct` layout in memory, sizes of `enums`, the sign of the `char` type, and so on are modelled to match the GNU C Compiler [99] for the 32-bit ARMv6 architecture.

3.4.1 Translation overview

The translation from C to Isabelle/HOL takes place in several stages:

1. The input C file is run through an external C preprocessor, which expands `#includes` and macros, and carries out other preprocessing directives;
2. The resulting C file is then parsed by Norrish's parser;
3. Isabelle/HOL types are generated for each `struct` used in the program;
4. Local and global variables are analysed to generate two new types: a *globals* record, which tracks global variables, and a *'a myvars* record, which tracks local variables;
5. Each individual function is translated into its equivalent Simpl representation; and finally,
6. Basic proofs are carried out on the generated functions, stating what variables the function modifies.

Figure 3.4 shows an example of a simple `max` function written in C and its corresponding representation generated by Norrish's C-to-Isabelle parser in Isabelle/HOL.

In the next sections, we look at three of the steps which are particularly relevant for the rest of this document: the generation of the `struct` types, generation of the *globals* and *'a myvars* records, and the conversion from C to Simpl.

3.4.2 Converting C types to Isabelle/HOL types

Before the C-to-Isabelle parser can emit function bodies for C functions, it must first declare appropriate Isabelle/HOL types for the program. Each basic C type, such as `int`, `unsigned short`, or `signed char` is mapped onto an Isabelle/HOL *word type*. Isabelle/HOL word types support an arbitrary number of bits, so *word32* represents a 32-bit word, *word8* an 8-bit word, *word7139* a 7139-bit word, and so on. We have also extended Isabelle/HOL to support signed word types; *sword32* represents a signed 32-bit word, for instance. We use the convention of suffixing unsigned word operations with "`w`" and signed word operations with "`s`"; so $a +_w b$ denotes unsigned addition, while $a +_s b$ denotes signed addition.[9]

Pointers in Isabelle/HOL are modelled using a custom pointer datatype, defined as

$$\textbf{datatype}\ 'a\ ptr = \mathsf{Ptr}\ word32$$

That is, a pointer is simply a *word32* representing the address being pointed to. The pointer type *'a ptr* has a phantom type parameter *'a* which is used to track the type of the

[9] As the signed word type models a two's-complement system, almost all operations on the unsigned word and signed word types are identical; the main point of difference are the signed division and signed modulo operations, which differ from their unsigned counterparts. For instance, the division of the unsigned value -1 is $2^{31} - 1$, while the division of the signed value -1 by 2 is simply 0.

Table 3.2: C types and their corresponding Isabelle/HOL types.

C type	Isabelle type
`signed char`	*sword8*
`unsigned short`	*word16*
`signed int`	*sword32*
`unsigned int *`	*word32 ptr*
`unsigned int **`	*word32 ptr ptr*
`struct node`	*node_C*

pointer; so a *word32 ptr* points to a *word32*, while a *sword8 ptr ptr* points to a *sword8 ptr*. The function ptr_val :: *'a ptr* ⇒ *word32* retrieves the value of a pointer, where

$$\text{ptr\_val}\ (\text{Ptr}\ a) = a$$

and the function ptr_coerce :: *'a ptr* ⇒ *'b ptr* allows a pointer to be coerced into a pointer of a different type, where

$$\text{ptr\_coerce}\ (\text{Ptr}\ p :: 'a\ ptr) = (\text{Ptr}\ p :: 'b\ ptr)$$

The C-to-Isabelle parser constructs an Isabelle/HOL record for each C `struct` declared in the input program, with the generated record having fields that match those of the C `struct`. For example, the C structure

```
struct node {
    int data;
    struct node *next;
};
```

would have the corresponding Isabelle/HOL record

record *node_C* =
  next_C :: *node_C* ⇒ *node_C ptr*
  data_C :: *node_C* ⇒ *sword32*

Here, the structure named `node` has been converted into a record of the same name suffixed with "`_C`". Each field of the structure has also been converted into a field of the record with the corresponding type, again suffixed with "`_C`". For each structure the C-to-Isabelle parser defines an Isabelle/HOL record for, it additionally generates definitions stating how the record can be encoded and decoded to and from raw bytes, along with proofs showing the consistency of these operations.

Table 3.2 provides a list of example C types and the equivalent Isabelle/HOL types they are translated into by the C-to-Isabelle parser.

3.4.3 Generation of state types

When converting a program from C into Isabelle/HOL, the C-to-Isabelle parser will generate an Isabelle/HOL record to track the global state of a program (such as global

variables and the contents of memory), and a record to track the local state of the program (such as local variables, function arguments and return values). These types are named *globals* and *'a myvars*, respectively, and are described in detail below.

Generating the globals record

The *globals* record, which represents all global program state, contains: (*i*) a field for each global variable in the program; (*ii*) a field named t_hrs_′ that represents the system's heap, discussed further in Chapter 7; and (*iii*) fields phantom_machine_state_′ for *machine state* and ghost′state_′ for *ghost state*, used to model hardware and other parts of the program not directly visible in the source code respectively.[10]

For example, the globals record for a simple program with three global variables, `int x`, `unsigned short y`, and `int *z` would be as follows:

record *globals* =
  t_hrs_′ :: (*word32* ⇒ *word8*) × (*word32* ⇒ *typ_tag*)
  phantom_machine_state_′ :: *unit*
  ghost′state_′ :: *unit*
  x_′ :: *sword32*
  y_′ :: *word16*
  z_′ :: *sword32 ptr*

Here, each global variable `g` in the program is given an associated field with a name of the form *g_′*.

Generating the locals record

The C-to-Isabelle parser additionally generates a record *'a myvars* which tracks the values of local variables. Each local variable `l` in the program is given a field named *l_′* in the record. As Simpl provides no mechanism to pass parameters into functions or return values out of functions, the *'a myvars* record also contains fields to track these parameters and return values. Parameters, like local variables, take the form *p_′* where `p` is the name of the parameter; while return values take the form ret__*T*_′, where *T* is the name of the type being returned.

The C-to-Isabelle parser uses Simpl exceptions to model abrupt termination commands, such as `break`, `continue`, and `return` statements. To help model these, the *'a myvars* record contains a field global_exn_var_′ used to store the reason for the current Simpl exception, which will contain one of the values Break, Return, or Continue.

Finally, the *'a myvars* record contains a field globals with type *'a* that contains the current global state of the program. In our work, we always instantiate *'a* to the *globals* type described above. That is, local variable data is stored in a record of type *globals myvars* while global variable data is stored in a record of type *globals* which is stored inside the field globals of the *globals myvars* record.

[10] We don't discuss ghost state or phantom machine state further in this document. Interested readers can refer to examples such as Daum et al.'s work [36], where it is used to model the current state of the CPU's MMU, for instance.

For example, the *globals myvars* record for the `max` program in Figure 3.4 is as follows:

record *globals myvars* =
  globals :: *globals*
  global_exn_var_′ :: *c_exntype*
  ret__int_′ :: *sword32*
  a_′ :: *sword32*
  b_′ :: *sword32*

If the program contained other functions, additional fields in this record would be present for the local variables in those functions, parameters of those functions, and return types of those functions.

Each function executes with its own copy of the *globals myvars* record.[11] This allows functions that share local variable names or recursive function calls to be modelled without problems. If two functions have the same local variable name with different types, name mangling is used to give them unique names in the *globals myvars* record.

Simpl notation

For both local variables and global variables, Simpl representations of programs often use the notation ‘x to represent the term x_′ *s*. In the former notation, the state variable *s* is implicit. For example, the condition of the `max` program in Figure 3.4

$$\text{Cond } \{\!|\text{`a} \leq_s \text{`b}|\!\} \ \ldots$$

would be written long-hand as

$$\text{Cond } \{s.\ \text{a\_}'\ s \leq_s \text{b\_}'\ s\} \ \ldots$$

In this document, we continue to use Schirmer's notation for Simpl code fragments, but avoid using it elsewhere.

3.4.4 Generation of Simpl

Once Isabelle/HOL types for the input program have been generated, the C-to-Isabelle parser is finally able to emit Isabelle/HOL Simpl definitions for each input C function.

Figure 3.4 shows an example of the function `max` parsed into Simpl. At the entry of the function, the values of parameters `a` and `b` are assumed to have been setup by the caller. The variable ret__int is used to track the return value of the function, which is then made available to the caller.

As mentioned previously, exceptions are used to model the `return` statement. In particular, the C `return` statement is translated into a sequence of Simpl commands that: (*i*) write the return value to the variable ret__int; (*ii*) set the variable global_exn_var

[11] In particular, function calls are modelled by the C-to-Isabelle parser as saving a copy of the current local variables and restoring them when the called function returns; this is analogous to the way the C compiler allocates a new stack frame for each function call.

to the value Return, indicating that the reason for the exception is a `return` statement; and finally (*iii*) uses the Simpl Throw statement to raise an exception. The exception is caught at the outside of the function body, where normal execution resumes. The variable global_exn_var is used by the C-to-Isabelle parser when there may be multiple reasons for an exception: such as a `return` statement inside a loop body that also contains a `break` statement. In this case, the exception triggered by the `break` statement needs to be caught at the end of the loop's body, while the exception triggered by the `return` statement needs to propagate to the end of the function body: the global_exn_var variable distinguishes between the different cases.

To ensure that the C program doesn't exhibit undefined behaviour (described in Section 3.1.2), the C-to-Isabelle parser emits Simpl Guard statements that ensure operations are safe. The translated `max` program has a single Guard statement that ensures that control flow doesn't drop off the end of the function without returning a value. The C-to-Isabelle parser will also emit Guard statements that ensure signed arithmetic operations don't overflow, that division operations do not divide by zero, that pointer dereferences are safe, and so on.

As can be seen in Figure 3.4, the final result of the C-to-Isabelle parser is rather verbose; much of this is due to the C-to-Isabelle parser's attempts to keep the output a literal translation of the input C, minimising any cleverness. In the rest of this document, we will look at how the resulting code can be drastically simplified, while simultaneously generating proofs that our simplified version formally corresponds to this original Simpl.

Chapter Summary

- The *C language*, despite its age, still remains in use in safety-critical, security-critical, and real-time programs today. This is in large part because of C's small size, predictability and efficiency. The goal of our work is to verify C programs.
- The *Isabelle/HOL interactive theorem prover* allows general mathematical reasoning, both by human users and by programs using its ML API. Isabelle/HOL's design—which uses a small LCF-style proof kernel internally—ensures that extensions to Isabelle/HOL are unable to introduce unsoundness.
- Schirmer's *Simpl language* is designed to allow imperative programs to be easily modelled and reasoned about in Isabelle/HOL. Simpl has deeply embedded statements, and shallowly embedded expressions.
- Norrish's *C-to-Isabelle parser* converts C into Simpl, allowing C code to be reasoned about using Isabelle/HOL. Norrish's parser generates a conservative representation of C, at the cost of its output being harder to reason about. Our work uses the output of the C-to-Isabelle parser as its starting point.

4 From deep to shallow embeddings

In this chapter, we describe the differences between programs represented in logic using a deep embedding and those using a shallow embedding. In the context of program verification, we argue that shallowly embedded representations are easier to reason about. Unfortunately, Norrish's conservative C-to-Isabelle parser generates deeply embedded representations of C.

The goal of this chapter is to develop techniques to automatically and verifiably translate the deeply-embedded Simpl programs generated by the C-to-Isabelle parser into a shallowly embedded representation. Section 4.2 describes Cock et al.'s existing monadic framework, which we use as our foundation for producing shallowly embedded program representations. Cock et al.'s framework does not support loops, however, so in Section 4.3 we formalise a new monadic *while-loop combinator*, allowing us to model imperative loops. We additionally develop rules to ease manual reasoning about our while-loop combinator.

With a suitable target representation now available, we next perform an automatic, proof-producing conversion from Schirmer's Simpl language into Cock et al.'s monadic framework. This conversion is described in Section 4.4. Finally, Section 4.5 describes program simplifications our new monadic representation enables us to carry out, significantly easing reasoning.

This chapter is based on the published work by Greenaway et al. [49], *Bridging the gap: automatic verified abstraction of C* in ITP 2012.

4.1 Reasoning in deep and shallow embeddings

Before we can begin to formally reason about a program, we must first translate it into the logic of our theorem prover. During this process, a decision must be made as to what logical representation we should use for our program. One choice to be

made when selecting a representation is whether we use a *deep embedding* or a *shallow embedding* [19]. In a deep embedding, the *structure* of the program is encoded into the logic, while in a shallow embedding the *semantics* of the program are encoded into the logic.

To better understand the differences between the two, consider the following equation:

$$2 + 2 = 4$$

Is the left-hand side equal to the right-hand side? The obvious answer is 'yes': the left-hand side has a semantic value of 4, and the right-hand side also has a semantic value of 4. So, clearly, the two sides are equal.

But we could instead argue that the two sides are not equal: the left-hand side is three characters long, while the right-hand side is only one character long. $3 \neq 1$, so clearly the two sides are not equal.[1]

In our first argument, we are concerned about the *semantics*, not the structure of the two sides: we are treating the expression as shallowly embedded. In our second argument, we are concerned about the *structure* of the two sides: the expression is deeply embedded.

While having a deep embedding for program statements is sufficient for reasoning about program behaviour, in practice it is frustrating. Because two semantically equivalent program fragments are only considered equal if they are *structurally* identical, many standard mechanisms in theorem provers cannot be used. For instance, term rewriting—which allows sub-terms of a program to be replaced with simpler, semantically equivalent alternatives—cannot be used on a deeply embedded representation of a program. While tools can be developed to alleviate some of this burden [110], much of the inbuilt reasoning support available in provers such as Isabelle/HOL remains unavailable. In contrast, when reasoning about a shallowly embedded representation of a program, users can freely switch back and forth among different semantically equivalent representations, using the most convenient representation at any point in time.

In some scenarios, a verification engineer may *need* to reason about a program's structure, however, in which case deeply embedded program representations become necessary. For instance, if we wanted to reason about the timing characteristics of assembly implementations of the above equations, then we would need to know the precise representation of the program. Such information is lost when switching to a shallow embedding. Alternatively, we might want to carry out meta-reasoning about a language, such as showing that programs written in a particular subset of the language satisfy a given property. Such reasoning would also require a deeply embedded representation of the language so that we can quantify over all possible programs in our proofs.

In this work, our aim is to reason about the *semantics* of programs written in C, and not their structure, so we would like to use a shallowly embedded representation if possible. Norrish's C-to-Isabelle parser, which is the starting point for our work, converts

[1] Typically a deep embedding would not use the representations "2 + 2" or "4", but instead something closer to "Add (Const 2) (Const 2)" and "Const 4" respectively, which more explicitly shows that we are encoding the expression's abstract syntax tree in our logic.

C into the Simpl language, which uses deeply embedded statements and shallowly embedded expressions.[2] In order to present a shallow embedding to the user we must find a suitable representation for low-level imperative programs and transform Simpl into this new representation.

4.2 Cock et al.'s monadic framework

Our goal is to represent imperative programs encoded in Schirmer's deeply-embedded *Simpl* language into a shallowly-embedded representation that eases reasoning. But which representation is best suited to such reasoning?

Any representation we choose must be at least as expressive as Simpl, being able to represent programs that read and write global state, contain loops that potentially do not terminate, raise and catch exceptions, contain non-determinism, and that may irrecoverably fail.

In this section we introduce *state monads* and describe how they can be used to model imperative programs in a functional language such as Isabelle/HOL. We next describe Cock et al.'s [30] existing formalisation of state monads in Isabelle/HOL, along with some existing tools that can assist reasoning about them.

4.2.1 Introducing the state monad

A common approach in functional programming languages is to use *state monads* to model imperative-style programs [104]; we use this same approach to model such functions in Isabelle.

To understand monads, we start by considering a naïve method of representing imperative programs that perform a sequence of calculations inside a functional let block:

$$\begin{aligned}&\mathsf{let}\ x = b * b - 4 * a * c;\\ &\quad\ y = -\ b + \mathsf{sqrt}\ x;\\ &\quad\ z = 2 * a\\ &\mathsf{in}\ y\ \mathsf{div}\ z\end{aligned}$$

In this program fragment, we carry out a calculation on input variables a, b and c. The temporary variables x, y, z are *bound variables*, holding intermediate calculations. The final value of the expression is the term y div z.

The let syntax above is nothing particularly magical, but is simply syntactic sugar for the equivalent lambda expression:

$$(\lambda x.\ (\lambda y.\ (\lambda z.\ y\ \mathsf{div}\ z)\ (2 * a))\ (-\ b + \mathsf{sqrt}\ x))\ (b * b - 4 * a * c)$$

Imperative programs often need to model *global state*, where variables are read and written to memory. While our previous model doesn't let us model such state, we can

[2] That is, expressions such as $2+2$ and 4 are encoded semantically and hence would be considered equivalent, but statements such as Skip and (Skip;; Skip) are encoded structurally and hence would be considered different, despite being semantically equivalent.

easily extend it by adding an additional state variable to each line that tracks the current global state. We additionally modify the program so that the variables a, b, and c are written out of this state, and the final value is written back to the variable a:

$$\begin{array}{l}\mathsf{let}\ (x,\ s) = (\mathsf{b\_}'\ s * \mathsf{b\_}'\ s - 4 * \mathsf{a\_}'\ s * \mathsf{c\_}'\ s,\ s);\\ \quad (y,\ s) = (-\ \mathsf{b\_}'\ s + \mathsf{sqrt}\ x,\ s);\\ \quad (z,\ s) = (2 * \mathsf{a\_}'\ s,\ s)\\ \mathsf{in}\ ((),\ s(\!|\mathsf{a\_}' := y\ \mathsf{div}\ z|\!))\end{array}$$

In this example, each line now calculates two values: a *return value*, which is bound to a temporary variable; and a new global state, which by convention we always bind to the variable *s*.[3] We hit a slight snag with the final line of the program: we want every statement in our program to calculate both a return value and a new state, but the final line doesn't have any useful return value—it simply updates the state. Our solution is to simply make it return the element unity element (written "()") of type *unit*, which we use to indicate that the return value is not significant.

Another aspect of imperative programs that we may wish to model is the concept of *irrecoverable failure*. For example, the argument to sqrt should not be negative, nor should the argument to the final divide operation be zero. If either occurs, we consider the program to have failed. To model this idea of irrecoverable failure, we can yet again tweak our program representation to maintain a *failure flag*, like so:

$$\begin{array}{l}\mathsf{let}\ (x,\ s,\ f) = (\mathsf{b\_}'\ s * \mathsf{b\_}'\ s - 4 * \mathsf{a\_}'\ s * \mathsf{c\_}'\ s,\ s,\ \mathsf{False});\\ \quad (y,\ s,\ f) = (-\ \mathsf{b\_}'\ s + \mathsf{sqrt}\ x,\ s,\ f \vee x < 0);\\ \quad (z,\ s,\ f) = (2 * \mathsf{a\_}'\ s,\ s,\ f)\\ \mathsf{in}\ ((),\ s(\!|\mathsf{a\_}' := y\ \mathsf{div}\ z|\!),\ f \vee z = 0)\end{array}$$

Each statement now generates three values: a return value, the updated state, and a failure flag. The failure flag is set to True if either the current statement fails, or if any previous statement has already failed. We can reason about program failure by inspecting the final value of this failure flag.

From here we could attempt to model non-determinism (for instance, by having a set of states generated by each statement, instead of just a single state), or abrupt termination (perhaps by adding a flag indicating if we are executing normally or not), but already everything has become rather unwieldy. Let's take a step back.

The alternative monadic approach to modelling imperative programs would be to modify statements to be *functions* that accept the current program state, and generate a return value and modified state. For example, our earlier example—before we attempted to model failure—could be represented as follows:

$$\begin{array}{l}\mathsf{let}\ (x,\ s) = (\lambda s.\ (\mathsf{b\_}'\ s * \mathsf{b\_}'\ s - 4 * \mathsf{a\_}'\ s * \mathsf{c\_}'\ s,\ s))\ s;\\ \quad (y,\ s) = (\lambda s.\ (-\ \mathsf{b\_}'\ s + \mathsf{sqrt}\ x,\ s))\ s;\\ \quad (z,\ s) = (\lambda s.\ (2 * \mathsf{a\_}'\ s,\ s))\ s\\ \mathsf{in}\ ((),\ s(\!|\mathsf{a\_}' := y\ \mathsf{div}\ z|\!))\end{array}$$

While this doesn't initially look like we have gained much, by moving to this higher-order

[3] We reuse the same bound variable name *s* through the function. Each time we reuse the name, the previous value of *s* is *shadowed*. A semantically equivalent approach would be to use unique names such as s, s', s'', and so on.

representation of states, we are able to factor out the 'glue' holding the statements together into its own function that accepts individual statements as arguments. This glue function is traditionally called *bind* and has the notation $f \gg\!= g$. For our previous example, an appropriate bind function is as follows:

$$\begin{aligned} a \gg\!= b \equiv \\ \quad \lambda s.\ b\ (\mathsf{fst}\ (a\ s))\ (\mathsf{snd}\ (a\ s)) \end{aligned}$$

With this, we can rewrite our function as follows:

$$\begin{aligned} &(\lambda s.\ (\mathsf{b\_}'\ s * \mathsf{b\_}'\ s - 4 * \mathsf{a\_}'\ s * \mathsf{c\_}'\ s, s)) \\ &\quad \gg\!= (\lambda x.\ (\lambda s.\ (-\ \mathsf{b\_}'\ s + \mathsf{sqrt}\ x, s)) \\ &\qquad \gg\!= (\lambda y.\ (\lambda s.\ (2 * \mathsf{a\_}'\ s, s)) \\ &\qquad\quad \gg\!= (\lambda z\ s.\ ((), s(\!|\mathsf{a\_}' := y\ \mathsf{div}\ z|\!))))) \end{aligned}$$

Next, we introduce two monadic functions, defined as follows:

$$\begin{aligned} \mathsf{gets}\ f &\equiv \lambda s.\ (f\ s, s) \\ \mathsf{modify}\ f &\equiv \lambda s.\ ((), f\ s) \end{aligned}$$

The function gets calculates a value based on the current state, while modify updates the state to a new value. More precisely, the term $\mathsf{gets}\ (\lambda s.\ \mathsf{a\_}'\ s + 2)$ is a function that takes the current state, and returns a bound value `a + 2` and a new state (which is defined to be the same as the input state). The modify function is similar, but instead of returning a useful bound variable (in particular, it simply returns the unit element $()$), it instead returns a modified state.

After rewriting our example to use the new functions, we now have:

$$\begin{aligned} &\mathsf{gets}\ (\lambda s.\ \mathsf{b\_}'\ s * \mathsf{b\_}'\ s - 4 * \mathsf{a\_}'\ s * \mathsf{c\_}'\ s) \\ &\quad \gg\!= (\lambda x.\ \mathsf{gets}\ (\lambda s.\ -\ \mathsf{b\_}'\ s + \mathsf{sqrt}\ x) \\ &\qquad \gg\!= (\lambda y.\ \mathsf{gets}\ (\lambda s.\ 2 * \mathsf{a\_}'\ s) \\ &\qquad\quad \gg\!= (\lambda z.\ \mathsf{modify}\ (\lambda s.\ s(\!|\mathsf{a\_}' := y\ \mathsf{div}\ z|\!))))) \end{aligned}$$

Finally, adding some syntactic sugar to the bind function, we arrive at:

$$\begin{aligned} &\textbf{do}\ x \leftarrow \mathsf{gets}\ (\lambda s.\ \mathsf{b\_}'\ s * \mathsf{b\_}'\ s - 4 * \mathsf{a\_}'\ s * \mathsf{c\_}'\ s); \\ &\quad y \leftarrow \mathsf{gets}\ (\lambda s.\ -\ \mathsf{b\_}'\ s + \mathsf{sqrt}\ x); \\ &\quad z \leftarrow \mathsf{gets}\ (\lambda s.\ 2 * \mathsf{a\_}'\ s); \\ &\quad \mathsf{modify}\ (\lambda s.\ s(\!|\mathsf{a\_}' := y\ \mathsf{div}\ z|\!)) \\ &\textbf{od} \end{aligned}$$

By hiding the plumbing behind definitions such as the bind operator, gets, and modify, we are able to increase the sophistication of our model without further complicating its representation. If we wanted to introduce the failure flag back into our model, we would need to update the definitions of bind, gets, and modify, but our program would be unchanged. We could similarly add support for exceptions (by making the bind operator ignore its right-hand side if an exception is currently active), or add support for non-determinism (by tweaking the bind operator to deal with sets).

In the following sections, we will describe an existing monadic formalisation by Cock et al. [30] that supports modelling global state, failure and non-determinism. We will later describe an extension of this monadic framework that adds support for raising and catching exceptions.

4.2.2 The state monad

In AutoCorres we primarily use two monads: a *state monad*, which tracks global state and additionally has support for non-determinism and failure; and an *exception monad*, which adds support for raising and catching exceptions.

We utilise the existing monadic framework of Cock, Klein, and Sewell [30], which provides definitions and theorems for the basic primitives for both the state and exception monad, as well as providing a verification-condition generator (VCG), which assists with Hoare-style reasoning on monadic programs.

Each monadic statement in Cock et al.'s state monad has the type

$$'state \Rightarrow ('ret \times 'state)\ set \times bool$$

which we abbreviate as $('state, 'ret)\ monad$.

Such functions accept a single input state $s :: 'state$ and return a tuple. The first half of this tuple contains the results of the execution: a set of pairs containing a return value and new state. The result is a set, so that functions may generate more than one state or return value to represent non-determinism. The second half of the tuple is the *failure flag* indicating whether any execution of the monad failed. If the flag is True then at least one failure has occurred, while False indicates that all executions have succeeded. Instead of using Isabelle's standard names fst and snd, we name the first element of this tuple results and the second element failed:

$$\mathsf{results}\ (a,\ b) = a$$
$$\mathsf{failed}\ (a,\ b) = b$$

The function return a, as the name suggests, simply returns the value a without modifying the state. It is defined as follows:

$$\mathsf{return}\ a \equiv \lambda s.\ (\{(a,\ s)\},\ \mathsf{False})$$

The function accepts the current state s, and returns a single result consisting of the unmodified state s and a value a.

The monadic bind operator for the state monad is defined as follows:

$$\begin{aligned} f >\!\!>\!= g \equiv\ & \\ & \lambda s.\ (\{(r'',\ s'').\ \exists (r',\ s') \in \mathsf{results}\ (f\ s).\ (r'',\ s'') \in \mathsf{results}\ (g\ r'\ s')\}, \\ & \quad \mathsf{failed}\ (f\ s) \vee (\exists (r',\ s') \in \mathsf{results}\ (f\ s).\ \mathsf{failed}\ (g\ r'\ s'))) \end{aligned}$$

While this definition is rather lacking in aesthetic beauty, there is nothing particularly deep going on in it. To obtain the results of $(f >\!\!>\!= g)\ s$, we simply consider every possible result (r', s') of $f\ s$, pass it into the function g, and then collate the results. Similarly, $(f >\!\!>\!= g)\ s$ fails if either $f\ s$ fails, or if $g\ r'\ s'$ fails for some output (r', s') of $f\ s$.

With these two definitions in place, Cock et al. prove that the three standard monad laws (left identity, right identity and associativity of bind) all hold:

$$(\mathsf{return}\ x >\!\!>\!= f) = (f\ x) \qquad \textsc{ReturnBind}$$

$$(f >\!\!>\!= \mathsf{return}) = (f) \qquad \textsc{BindReturn}$$

$$((f >\!\!>\!= g) >\!\!>\!= h) = (f >\!\!>\!= (\lambda x.\ g\ x >\!\!>\!= h)) \qquad \textsc{BindAssoc}$$

State Monad	Simpl	Definition
skip	Skip	$\lambda s.\ (\{((), s)\}, \text{False})$
modify m	Basic m	$\lambda s.\ (\{((), f\ s)\}, \text{False})$
condition $c\ L\ R$	Cond $c\ L\ R$	$\lambda s.$ if $P\ s$ then $L\ s$ else $R\ s$
spec r	Spec r	$\lambda s.\ (\{()\} \times \{s'.\ (s, s') \in R\}, \nexists s'.\ (s, s') \in R)$
select S	—	$\lambda s.\ (A \times \{s\}, \text{False})$
guard P	Guard $z\ C$ Skip	$\lambda s.\ (\text{if } P\ s \text{ then } \{((), s)\} \text{ else } \emptyset, \neg\ P\ s)$
fail	Guard $z\ \emptyset$ Skip	$\lambda s.\ (\emptyset, \text{True})$
return v	—	$\lambda s.\ (\{(a, s)\}, \text{False})$
gets f	—	$\lambda s.\ (\{(f\ s, s)\}, \text{False})$

Table 4.1: Monadic functions and their equivalent in the Simpl language.

The simple state monad is sufficient to allow us to define monadic equivalents for the simpler statements in the Simpl language. These are shown in Table 4.1, and include functions to update the global state (modify), conditionally execute instructions (condition) and to model non-determinism (spec and select). One minor difference between Simpl and our monadic representations is that Simpl uses sets for its conditions and guards ($s \in C$), while we use predicates ($P\ s$), which we find more natural to work with.

Cock et al. also introduce a new monadic command fail representing unconditional program failure, which we will find useful in later sections. The fail statement has no direct equivalent in Simpl, but could be modelled by Guard $z\ \emptyset$ Skip, which unconditionally enters the state Fault z.

4.2.3 Reasoning about the state monad

Being able to *specify* imperative functions in a monadic style is of no use if we don't have any way to *reason* about their behaviours. To assist with reasoning about monadic programs, Cock et al. developed a framework allowing Hoare triples to be written about monadic programs, along with tools to assist with reasoning about these specifications.

First, a predicate named valid is defined as follows:

$$\begin{aligned}&\{\!|\,P\,|\!\}\ f\ \{\!|\,Q\,|\!\} \equiv \\ &\qquad \forall s.\ P\ s \longrightarrow (\forall (r', s') \in \text{results}\ (f\ s).\ Q\ r'\ s')\end{aligned}$$

Intuitively, this states that if the precondition P holds on a state s, then the postcondition Q will also hold on all results of executing f on s. P has the type $'s \Rightarrow bool$, taking the initial program state. Q has the type $'r \Rightarrow 's \Rightarrow bool$, taking both the return value of the monad f and the resulting state.

The definition allows us to write Hoare triples that reason about the return value of the monad, such as

$$\{\!|\,\lambda s.\ \text{True}\,|\!\}\ \text{return}\ (2 + 2)\ \{\!|\,\lambda rv\ s.\ rv = 4\,|\!\}$$

which declares that the return value of the monadic statement return $(2 + 2)$ is 4. Additionally, we can write Hoare triples that refer to both the input state and the final

state of a monadic program. For instance, in a program whose global state consists solely of a single natural number, we could write

$$\{\!|\lambda s.\ s = 3|\!\}\ \mathsf{modify}\ (\lambda s.\ s + 2)\ \{\!|\lambda rv\ s.\ s = 5|\!\}$$

This statement declares that, assuming the state starts with the value 3, if we execute modify $(\lambda s.\ s + 2)$, then the final value of the state will be 5.

Frequently, we want to know not only that a given monad fulfils a given postcondition, but also that it does so without failing. We have extended Cock et al.'s Hoare logic framework, introducing a second predicate $\mathsf{valid}_{\mathsf{NF}}$ (where the suffix "NF" indicates "no failure"). Our definition is similar to Cock et al.'s valid definition, but ensures that the failure flag is clear at the end of execution:

$$\begin{aligned}&\{P\}\ f\ \{Q\} \equiv \\ &\qquad \{\!|P|\!\}\ f\ \{\!|Q|\!\} \wedge (\forall s.\ P\ s \longrightarrow \neg\ \mathsf{failed}\ (f\ s))\end{aligned}$$

The difference between the notations "$\{\!|P|\!\}\ f\ \{\!|Q|\!\}$" and "$\{P\}\ f\ \{Q\}$" can be remembered as follows: the hollow brackets give hollow guarantees about failure, while the solid brackets give solid guarantees.

In practice, the main differences that appear when reasoning about the two are as follows:

- In the valid framework, guard statements may be assumed correct, while in $\mathsf{valid}_{\mathsf{NF}}$ they must be proven to hold;
- In the valid framework, it can be assumed that fail statements are unreachable, while in the $\mathsf{valid}_{\mathsf{NF}}$ framework, they must be proven to be unreachable;
- In the $\mathsf{valid}_{\mathsf{NF}}$ framework, the user has an additional obligation to show that every executed spec statement has at least one output.

Thus, the following statement with an invalid guard is true for valid:

$$\{\!|\lambda s.\ \mathsf{True}|\!\}\ \mathsf{guard}\ (\lambda s.\ 2 + 2 = 5)\ \{\!|\lambda rv\ s.\ \mathsf{True}|\!\}$$

but is not true for $\mathsf{valid}_{\mathsf{NF}}$:

$$\neg\ (\{\lambda s.\ \mathsf{True}\}\ \mathsf{guard}\ (\lambda s.\ 2 + 2 = 5)\ \{\lambda rv\ s.\ \mathsf{True}\})$$

To actually reason about such Hoare triples, Cock et al. also developed a calculus that allows the weakest precondition to be calculated for a given postcondition.[4] The calculation takes place using an Isabelle tactic named '`wp`' developed by Cock et al. The tactic will apply a set of syntax-directed rules, typically leaving the user to prove a single implication that the Hoare triple's precondition implies the calculated weakest-precondition.

[4] In practice, the user-extensible calculus makes no guarantee that the calculated "weakest" precondition is in fact the weakest, but simply that it is *sufficient* to prove the given postcondition. If the user adds custom rules that do not strictly generate the weakest precondition, but just a 'weakish' one, the framework may generate goals which cannot be proven, but no unsoundness will be introduced.

$$\{\!|P\ ()|\!\}\ \mathsf{skip}\ \{\!|P|\!\}\ \textsc{SkipWp} \qquad \{\!|P\ x|\!\}\ \mathsf{return}\ x\ \{\!|P|\!\}\ \textsc{ReturnWp} \qquad \{\!|\lambda s.\ P\ (f\ s)\ s|\!\}\ \mathsf{gets}\ f\ \{\!|P|\!\}\ \textsc{GetsWp}$$

$$\{\!|\lambda s.\ P\ ()\ (f\ s)|\!\}\ \mathsf{modify}\ f\ \{\!|P|\!\}\ \textsc{ModifyWp} \qquad \{\!|\lambda s.\ \forall x \in S.\ Q\ x\ s|\!\}\ \mathsf{select}\ S\ \{\!|Q|\!\}\ \textsc{SelectWp}$$

$$\frac{\{\!|Q|\!\}\ A\ \{\!|P|\!\} \qquad \{\!|R|\!\}\ B\ \{\!|P|\!\}}{\{\!|\lambda s.\ \mathsf{if}\ C\ s\ \mathsf{then}\ Q\ s\ \mathsf{else}\ R\ s|\!\}\ \mathsf{condition}\ C\ A\ B\ \{\!|P|\!\}}\ \textsc{ConditionWp}$$

$$\{\!|\lambda x.\ \mathsf{True}|\!\}\ \mathsf{fail}\ \{\!|Q|\!\}\ \textsc{FailWp} \qquad \{\!|\lambda s.\ \forall t.\ (s,\ t) \in f \longrightarrow P\ ()\ t|\!\}\ \mathsf{spec}\ f\ \{\!|P|\!\}\ \textsc{SpecWp}$$

$$\frac{\forall x.\ \{\!|B\ x|\!\}\ g\ x\ \{\!|C|\!\} \qquad \{\!|A|\!\}\ f\ \{\!|B|\!\}}{\{\!|A|\!\}\ f \gg\!= g\ \{\!|C|\!\}}\ \textsc{BindWp} \qquad \frac{\{\!|Q|\!\}\ a\ \{\!|R|\!\} \qquad \forall s.\ P\ s \longrightarrow Q\ s}{\{\!|P|\!\}\ a\ \{\!|R|\!\}}\ \textsc{WeakenPre}$$

Table 4.2: A sample of backwards reasoning rules used to reason about Hoare-style valid triples.

A representative subset of the rules are shown in Table 4.2. We have developed an analogous set of rules for our own $\mathsf{valid}_{\mathsf{NF}}$ predicate, which are shown in Table 4.3.

For example, starting with the goal from earlier:

$$\{\!|\lambda s.\ s = 3|\!\}\ \mathsf{modify}\ (\lambda s.\ s + 2)\ \{\!|\lambda rv\ s.\ s = 5|\!\}$$

running the `wp` tactic automatically invokes the rules WeakenPre and ModifyWp, resulting in the subgoal:

```
1. ⋀s. s = 3 ⟹ s + 2 = 5
```

which is solved automatically by Isabelle/HOL's simplifier.

One final predicate, named $\mathsf{valid}_{\mathsf{EX}}$, allows us to state that there exists at least one path that satisfies a given postcondition. $\mathsf{valid}_{\mathsf{EX}}$ is defined as follows:

$$\begin{aligned}&\{\!|P|\!\}\ f\ \exists\{\!|Q|\!\} \equiv \\ &\qquad \forall s.\ P\ s \longrightarrow (\exists (rv,\ s') \in \mathsf{results}\ (f\ s).\ Q\ rv\ s')\end{aligned}$$

In the absence of non-determinism, valid and $\mathsf{valid}_{\mathsf{EX}}$ are equivalent. When non-determinism is present, however, we can use $\mathsf{valid}_{\mathsf{EX}}$ as a building block to show that a set of final states can actually be reached.

4.2.4 Modelling abrupt termination

Programs written in the imperative languages frequently use *abrupt termination*, where the control flow of the program transfers abruptly to a different location in the program. The most recognisable form of abrupt termination is in languages that feature exceptions, such as Java or C#. When an exception is thrown, execution transfers to the first handler

$$\frac{}{\{P\ ()\}\ \mathsf{skip}\ \{P\}}\ \textsc{SkipNf} \qquad \frac{}{\{P\ x\}\ \mathsf{return}\ x\ \{P\}}\ \textsc{ReturnNf} \qquad \frac{}{\{\lambda s.\ P\ (f\ s)\ s\}\ \mathsf{gets}\ f\ \{P\}}\ \textsc{GetsNf}$$

$$\frac{}{\{\lambda s.\ P\ ()\ (f\ s)\}\ \mathsf{modify}\ f\ \{P\}}\ \textsc{ModifyNf} \qquad \frac{}{\{\lambda s.\ \forall x \in S.\ Q\ x\ s\}\ \mathsf{select}\ S\ \{Q\}}\ \textsc{SelectNf}$$

$$\frac{\{Q\}\ A\ \{P\} \quad \{R\}\ B\ \{P\}}{\{\lambda s.\ \mathsf{if}\ C\ s\ \mathsf{then}\ Q\ s\ \mathsf{else}\ R\ s\}\ \mathsf{condition}\ C\ A\ B\ \{P\}}\ \textsc{ConditionNf}$$

$$\frac{}{\{\lambda s.\ \mathsf{False}\}\ \mathsf{fail}\ \{Q\}}\ \textsc{FailNf} \qquad \frac{}{\{\lambda s.\ (\forall t.\ (s,\ t) \in f \longrightarrow P\ ()\ t) \wedge (\exists t.\ (s,\ t) \in f)\}\ \mathsf{spec}\ f\ \{P\}}\ \textsc{SpecNf}$$

$$\frac{\forall x.\ \{B\ x\}\ g\ x\ \{C\} \quad \{A\}\ f\ \{B\}}{\{A\}\ f \gg= g\ \{C\}}\ \textsc{BindNf} \qquad \frac{\{Q\}\ a\ \{R\} \quad \forall s.\ P\ s \longrightarrow Q\ s}{\{P\}\ a\ \{R\}}\ \textsc{WeakenPreNf}$$

Table 4.3: A sample of weakest precondition rules used to reason about Hoare-style $\mathsf{valid}_{\mathsf{NF}}$ triples.

able to process the exception. While the C programming language doesn't have exceptions, it has `return`, `continue` and `break` statements which transfer control flow to the end of the function, end of the loop body, and outside the loop body, respectively.[5]

The Simpl language allows such constructs to be modelled using its Throw and Catch statements. For example C's `return` statement may be modelled in Simpl by wrapping the function body with a Catch command, and then replacing instances of the `return` statement with Throw. When the Throw is encountered, control flow will be transferred to the handler of the outer Catch statement.

Simpl's Throw/Catch mechanisms cannot be easily modelled in Cock et al.'s state monad presented above. We can instead use an extension to the state monad, also developed by Cock et al., that adds support for modelling exceptions. This new monad, named the *exception monad*, is built using the standard state monad by using a return type of the form $'e + 'r$. This is Isabelle/HOL's *sum type*, which can either have the 'right' value Inr $(r :: 'r)$, used to represent the monad returning normally with the value r; or the 'left' value Inl $(e :: 'e)$, used to represent the monad returning abruptly with the exception value e. We give these two values the notation Norm r and Exc e respectively.

To model functions in this new exception monad, Cock et al. defined a new set of monadic primitives, including a new return function and bind function. In our work, we use the convention of naming function using the exception monad with the suffix "E". For example, the exception monad equivalent of return is named $\mathsf{return}_{\mathsf{E}}$.

[5] C also includes the `goto` statement and the ill-conceived `setjmp` and `longjmp` functions. These statements cannot be readily modelled in Simpl, and thus are not supported by Norrish's C-to-Isabelle parser, nor in our own work.

Cock et al. define $\mathsf{return}_\mathsf{E}$ and the corresponding function $\mathsf{throw}_\mathsf{E}$ as follows:

$$\mathsf{return}_\mathsf{E}\ r \equiv \mathsf{return}\ (\mathsf{Norm}\ r)$$
$$\mathsf{throw}_\mathsf{E}\ e \equiv \mathsf{return}\ (\mathsf{Exc}\ e)$$

The definition of the exception monad's bind function is similarly built upon the state monad's bind function. To evaluate $f \gg\!=_\mathsf{E} g$, we first execute f. If the result is an exception (i.e., Exc e), we skip g and simply propagate the exceptional value. We can similarly introduce an exception-handler operator $\mathsf{catch}_\mathsf{E}\ f\ g$, which performs the symmetric operation: if the result of f is an exception, we execute g. Otherwise, we propagate the return value on. The two operations are defined as:

$$f \gg\!=_\mathsf{E} g \equiv$$
$$\quad \textbf{do}\ x \leftarrow f;\ \mathsf{case}\ x\ \mathsf{of}\ \mathsf{Exc}\ x \Rightarrow \mathsf{throw}_\mathsf{E}\ x \mid \mathsf{Norm}\ x \Rightarrow g\ x\ \textbf{od}$$

$$\mathsf{catch}_\mathsf{E}\ f\ g \equiv$$
$$\quad \textbf{do}\ x \leftarrow f;\ \mathsf{case}\ x\ \mathsf{of}\ \mathsf{Exc}\ x \Rightarrow g\ x \mid \mathsf{Norm}\ x \Rightarrow \mathsf{return}_\mathsf{E}\ x\ \textbf{od}$$

Functions defined to use the standard state monad can be *lifted* into the exception monad. We simply take the return value r of the monad and indicate that the value is non-exceptional by transforming it into Norm r:

$$\mathsf{lift}_\mathsf{E}\ f \equiv$$
$$\quad \textbf{do}\ r \leftarrow f;\ \mathsf{return}\ (\mathsf{Norm}\ r)\ \textbf{od}$$

Using lift_E, we can simply define exception-monad versions of our monadic operators by lifting the standard state monad definition. For instance, $\mathsf{modify}_\mathsf{E}$ is defined in terms of modify as follows:

$$\mathsf{modify}_\mathsf{E}\ m \equiv \mathsf{lift}_\mathsf{E}\ (\mathsf{modify}\ m)$$

Table 4.4 shows the full list of operations.

4.2.5 Reasoning about the exception monad

Now that we have monadic functions allowing us to model imperative functions that use exceptions, we finally need a way to be able to reason about such programs. Cock et al. extend their Hoare-logic framework to allow reasoning about exception monads like so:

$$\{\!|P|\!\}\ f\ \{\!|Q|\!\},\ \{\!|E|\!\} \equiv$$
$$\quad \{\!|P|\!\}\ f\ \{\!|\lambda v\ s.\ \mathsf{case}\ v\ \mathsf{of}\ \mathsf{Exc}\ e \Rightarrow E\ e\ s \mid \mathsf{Norm}\ r \Rightarrow Q\ r\ s|\!\}$$

$$\{\,P\,\}\ f\ \{\,Q\,\},\ \{\,E\,\} \equiv$$
$$\quad \{\,P\,\}\ f\ \{\,\lambda v\ s.\ \mathsf{case}\ v\ \mathsf{of}\ \mathsf{Exc}\ e \Rightarrow E\ e\ s \mid \mathsf{Norm}\ r \Rightarrow Q\ r\ s\,\}$$

Intuitively, $\{\!|P|\!\}\ f\ \{\!|Q|\!\},\ \{\!|E|\!\}$ states that, assuming the precondition P is true when f is executed, then (i) if the result of f is non-exceptional with value r and state s, then $Q\ r\ s$ will hold; or (ii) if the result of f is the exception with value e and state s, then $E\ e\ s$ will hold. The predicate $\{\,P\,\}\ f\ \{\,Q\,\},\ \{\,E\,\}$ is analogous, but also requires that no execution path fails. We continue to call these predicates *Hoare triples*, wilfully ignoring the increasing inaccuracy of the name.

Exception Monad	Simpl	Definition
$\text{return}_\text{E}\ v$	—	return (Norm v)
$\text{throw}_\text{E}\ e$	Throw	return (Exc e)
$\text{gets}_\text{E}\ f$	—	lift_E (gets g)
skip_E	Skip	lift_E skip
$\text{modify}_\text{E}\ m$	Basic m	lift_E (modify m)
$\text{spec}_\text{E}\ r$	Spec r	lift_E (spec r)
$\text{guard}_\text{E}\ P$	Guard z $\{s.\ P\ s\}$ Skip	lift_E (guard P)
fail_E	Guard z $\emptyset$ Skip	$\lambda s.$ ($\emptyset$, True)
$L \gg\!=_\text{E} R$	L;; R	*(see definition on Page 47)*
$\text{catch}_\text{E}\ L\ R$	Catch $L\ R$	*(see definition on Page 47)*
$\text{condition}_\text{E}\ c\ L\ R$	Cond $\{s.\ c\ s\}$ $L\ R$	$\lambda s.$ if $P\ s$ then $L\ s$ else $R\ s$
$\text{whileLoop}_\text{E}\ C\ B\ i$	While $e\ c$	*(see definition on Page 51)*

Table 4.4: Exception monad functions and their closest equivalent in the Simpl language.

For example, the following predicate states that the following simple program never returns normally (i.e., only the exception postcondition is evaluated) and also doesn't modify global state (i.e., the initial state matches the final state, because the modify_E statement is never reached):

$$\begin{array}{l}\{\lambda s.\ s = s_o\} \\ \quad \textbf{do}\ \text{throw}_\text{E}\ 42; \\ \qquad \text{modify}_\text{E}\ (\lambda s.\ s + 1) \\ \quad \textbf{od} \\ \{\lambda rv\ s.\ \text{False}\},\ \{\lambda rv\ s.\ rv = 42 \wedge s = s_o\}\end{array}$$

The goal can be proven simply by running Cock et al.'s `wp` tactic instantiated with a set of rules for the exception monad, analogous to state monad rules described in Section 4.2.3.

4.3 Extending Cock et al.'s monadic framework with imperative-style loops

While Cock et al.'s monadic framework contains most of the features we need to model imperative languages in Isabelle/HOL, there is one notable omission: a mechanism for modelling `while` loops.

This was not a problem in the context of Cock et al.'s work, as they used recursion to model loops in their imperative programs. In the context of AutoCorres—where our desire is to automatically translate Simpl language programs into monadic form—attempting to model loops using recursion would present two main problems:

1. Isabelle/HOL is a logic of total functions. In particular, before a recursive function can be defined in Isabelle/HOL, it must first be shown to terminate. While in

simple cases Isabelle/HOL's function package [60] can automatically prove termination [22], in more complex cases the user must manually prove termination themselves.

This presents a difficulty, as we have no way to automatically prove termination for non-trivial loops, and hence would struggle to automatically translate `while` loops into a recursive form. Even if we *could* automatically prove termination, we would still have a problem that some loops only terminate under certain circumstances: a simple loop that traverses a linked list will only terminate if the list doesn't loop back on itself.

2. Ignoring the above problems, modelling loops using recursion requires us to significantly modify the layout of programs that we are representing, with each loop in the program needing to be extracted into its own function. While this isn't a technical problem *per se*, end-users would find it more difficult to see the connection between the input C program and the output of our tool.

Instead, we define a new monadic *while-loop combinator* for both the state monad and exception monad to model imperative `while` loops. The simpler state monad whileLoop combinator has the type

$$('r \Rightarrow 's \Rightarrow bool) \Rightarrow ('r \Rightarrow ('s, 'r)\ monad) \Rightarrow 'r \Rightarrow ('s, 'r)\ monad$$

In particular, whileLoop $C\ B\ i$ constructs a loop from a condition C, a body B, and an initial *loop iterator* value i. While ever the condition C remains true, the loop body will be executed, passing in the current loop iterator value as an argument. The return value from executing one iteration of the loop body will become the loop iterator's value in the next loop iteration. When the loop condition finally becomes false, the final value of the loop iterator will be used as the whileLoop's return value. The loop iterator allows us to bind variables in one iteration of the loop, and use them both in the next iteration of the loop and after the loop completes. Multiple variables may be passed through the loop iterator by packing them into a tuple.

For example, the following program demonstrates a simple loop. The program's global state consists of a single natural number. The loop has a single variable r passed through its loop iterator, which is initialised to zero:

$$\begin{array}{l}\mathsf{whileLoop}\ (\lambda r\ s.\ r < 10)\\ \quad (\lambda r.\ \textbf{do}\ \mathsf{modify}\ (\lambda s.\ s * 2);\\ \qquad\quad \mathsf{return}\ (r + 1)\\ \qquad \textbf{od})\ 0\end{array}$$

The loop body will execute 10 times, with each iteration doubling the program's global state and increasing the loop iterator r by 1. Once r reaches 10, the loop's condition will become false and the loop will terminate.

Our formal definition of whileLoop requires two components. The first component is an inductively defined set whileLoop_results, which describes the possible execution paths through the loop. The second component is an inductively defined predicate whileLoop_terminates, which determines if the loop will terminate for a given input state.

We say that the set whileLoop_results $C\ B$ contains an element (a, b) if there is an execution path through the loop body B with condition C, that starts from state a and terminates in state b. Both a and b have the type $('r \times 's)\ option$; Some (r, s) represents the monad's state having value s and the loop iterator having the value r, while None represents a state where the loop's body B has failed.

Formally, whileLoop_results is inductively defined by the following 3 rules:

$$\frac{\neg\ C\ r\ s}{(\mathsf{Some}\ (r,\ s),\ \mathsf{Some}\ (r,\ s)) \in \mathsf{whileLoop\_results}\ C\ B}$$

$$\frac{C\ r\ s \qquad (r',\ s') \in \mathsf{results}\ (B\ r\ s) \qquad (\mathsf{Some}\ (r',\ s'),\ z) \in \mathsf{whileLoop\_results}\ C\ B}{(\mathsf{Some}\ (r,\ s),\ z) \in \mathsf{whileLoop\_results}\ C\ B}$$

$$\frac{C\ r\ s \qquad \mathsf{failed}\ (B\ r\ s)}{(\mathsf{Some}\ (r,\ s),\ \mathsf{None}) \in \mathsf{whileLoop\_results}\ C\ B}$$

The rules declare that (i) there is a path through the loop from a state to itself if the loop's condition C is false; (ii) if there is a valid path (y, z), and starting the loop in state x will lead to state y, then (x, z) is also a path through the loop; and finally (iii) if a particular state x causes the loop's body B to fail, then we say there is a path from x to the failure state None.

While we can use the whileLoop_results set to determine what results are produced in a finite number of steps, we are also interested in knowing whether there are infinite traces through the loop—that is, whether the loop may have non-terminating paths through it. We introduce an inductively defined predicate whileLoop_terminates $C\ B\ r\ s$ to determine if every execution through the loop with body B and condition C starting from state s and iterator r terminates. The predicate whileLoop_terminates is defined inductively using the following two rules:

$$\frac{\neg\ C\ r\ s}{\mathsf{whileLoop\_terminates}\ C\ B\ r\ s}$$

$$\frac{C\ r\ s \qquad \forall (r',\ s') \in \mathsf{results}\ (B\ r\ s).\ \mathsf{whileLoop\_terminates}\ C\ B\ r'\ s'}{\mathsf{whileLoop\_terminates}\ C\ B\ r\ s}$$

That is, any state where the condition $C\ r\ s$ is false terminates. States where the condition $C\ r\ s$ is true terminate if all paths leading from that state also terminate.

With these two definitions in place, we can finally define the while-loop combinator whileLoop:

$$\begin{aligned}&\mathsf{whileLoop}\ C\ B\ r \equiv\\ &\quad \lambda s.\ (\{(r',\ s').\ (\mathsf{Some}\ (r,\ s),\ \mathsf{Some}\ (r',\ s')) \in \mathsf{whileLoop\_results}\ C\ B\},\\ &\qquad (\mathsf{Some}\ (r,\ s),\ \mathsf{None}) \in \mathsf{whileLoop\_results}\ C\ B \vee\\ &\qquad \neg\ \mathsf{whileLoop\_terminates}\ C\ B\ r\ s)\end{aligned}$$

The combinator returns the set of all results that can be reached in a finite number of loop iterations from the input state, as calculated by whileLoop_results. Additionally, we say that the loop fails if any computation inside the loop fails or if there are any non-terminating paths through the loop.

We also create an instance of the while-loop combinator for the exception monad by using the loop iterator to determine if an exception has been raised, and modifying the loop's condition to detect this as follows:

$$\begin{aligned}&\mathsf{whileLoop}_{\mathsf{E}}\ C\ B\ r \equiv \\ &\quad \mathsf{whileLoop}\ (\lambda r\ s.\ \mathsf{case}\ r\ \mathsf{of}\ \mathsf{Exc}\ a \Rightarrow \mathsf{False} \mid \mathsf{Norm}\ v \Rightarrow C\ v\ s) \\ &\quad\quad (\lambda r\ s.\ \mathsf{case}\ r\ \mathsf{of}\ \mathsf{Norm}\ v \Rightarrow B\ v\ s)\ (\mathsf{Norm}\ r)\end{aligned}$$

In both these definitions, non-termination is indicated by setting the failure flag. Hypothetically, if we separately tracked non-termination and failure in our monad, we could reason about monadic failure and non-termination of the loop independently. As it is, we cannot distinguish between the two. Our reason for mapping non-termination onto the failure flag is pragmatic: using the above definition allows us to reuse large parts of Cock et al.'s existing monadic formalisation, and hence also reuse the large library of existing proofs and tools associated with it. In our experience, we have found that we typically want to prove both total correctness and termination about generated functions, so that inability to distinguish between the two types of errors isn't problematic in practice.

4.3.1 Reasoning about the while-loop combinator

So far, we have mostly omitted proofs about the monadic functions defined above. This is because the theorems developed thus far can be proven simply by unfolding the relevant definitions and simplifying the results. The while-loop combinator is different, however, as it is defined in terms of the inductive set whileLoop_results and inductive predicate whileLoop_terminates. Reasoning about the while-loop combinator using these raw definitions is tedious, as all but the most simple proofs requires at least three inductive steps: one for each of the two references to whileLoop_results in the definition, and another for the whileLoop_terminates reference.

In this section, we develop some basic rules to assist with manual reasoning about the while-loop combinator, and then develop further rules that can be used to carry out Hoare-style reasoning with the while-loop combinator. We have also created versions of these rules that apply to the exception while-loop combinator $\mathsf{whileLoop}_{\mathsf{E}}$, but for brevity we omit these rules and proofs.

Basic reasoning

Perhaps the most basic operation that can be carried out on a while-loop is *unrolling* the loop by one step. We can show that this is the case as follows:

Lemma 4.1 (Unroll whileLoop) A while-loop may be unrolled into a single execution of its body, followed by the remainder of the loop:

$$\begin{aligned}&\mathsf{whileLoop}\ C\ B\ r = \\ &\quad \mathsf{condition}\ (C\ r)\ (B\ r \gg\!= \mathsf{whileLoop}\ C\ B)\ (\mathsf{return}\ r)\end{aligned}$$

PROOF The proof proceeds by unfolding the definition of whileLoop, considering the two possible cases of $C\ r\ s$, and then using the definitions of whileLoop_results and bind_E to show the equality holds.

The rule WHILELOOPUNROLL lets us reason about loops with a fixed, small number of iterations, but can not be used for reasoning about loops with an unknown number of iterations. In order to generalise such reasoning, the traditional approach is to develop an *invariant* for the loop, which states what is true about every execution of the loop. If we can prove that (i) the invariant holds initially; and (ii) the invariant continues to hold each loop iteration; we can then assume that the invariant holds when the loop terminates. We formalise this idea for our while-loop combinator as follows:

Theorem 4.2 (WHILELOOPWP) Assuming that (i) whenever a loop's body B executes on a state satisfying invariant I and loop condition C, it maintains I; and (ii) the invariant I holds when the loop is entered into; then it will also hold when the loop finishes:

$$\frac{\forall r.\ \{\!|\lambda s.\ I\ r\ s \wedge C\ r\ s|\!\}\ B\ r\ \{\!|I|\!\}}{\{\!|I\ r|\!\}\ \mathsf{whileLoop}\ C\ B\ r\ \{\!|\lambda r\ s.\ \neg\ C\ r\ s \wedge I\ r\ s|\!\}}$$

PROOF We start by unfolding the definitions of valid and whileLoop. The proof proceeds by induction by using the rules of the whileLoop_results definition.

The above rule provides a basic mechanism for Hoare-style reasoning about loops when failure and termination need not be considered. One minor complication with the rule is that it can not be automatically applied by Cock et al.'s VCG "`wp`" tactic described earlier in Section 4.2.3, because the rule requires an invariant I to be manually specified. If the user wished to use the `wp` tactic on a program containing a loop, they would have to: (i) run the tactic `wp` to calculate the weakest precondition up to the point of the loop; (ii) manually apply the above rule with a specified invariant I; and (iii) continue to run `wp` on the resulting Hoare triples produced by the rule.

We can simplify this by introducing a new constant whileLoop_inv defined as follows:

$$\mathsf{whileLoop\_inv}\ C\ B\ r\ I\ R \equiv \mathsf{whileLoop}\ C\ B\ r$$

This simple definition contains two dummy parameters: an invariant parameter I and a termination relation R.[6] A user can manually annotate the loop with these additional parameters by using Isabelle/HOL's term rewriting features to convert whileLoop statements to whileLoop_inv statements with an appropriate instantiation of I and R. Once

[6] We will ignore this second parameter for the moment, and return to it below when we wish to reason about loop termination.

the loops have been annotated, automated tactics such as `wp` can then run over the commands uninterrupted using the rule

$$\frac{\begin{array}{c}\forall r.\ \{\!|\lambda s.\ I\ r\ s \wedge C\ r\ s|\!\}\ B\ r\ \{\!|I|\!\} \\ \forall r\ s.\ I\ r\ s \longrightarrow \neg\ C\ r\ s \longrightarrow Q\ r\ s\end{array}}{\{\!|I\ r|\!\}\ \mathsf{whileLoop\_inv}\ C\ B\ r\ I\ M\ \{\!|Q|\!\}}$$

The definition of whileLoop_inv does not require the annotations I or R to be correct—or even make any sense. The user must still prove that the loop maintains the invariant, and that the invariant is strong enough to prove whatever property is of interest. The whileLoop_inv parameters simply provide hints to automated tools as to what invariant should be used.

Hoare-style reasoning about termination and non-failure

The rules defined thus far have allowed us to reason about partial correctness, but have ignored the issue of both termination and program failure. In this section we develop rules allowing us to reason about these.

We start by introducing a rule that will allow us to prove properties about the predicate whileLoop_terminates $C\ B\ r\ s$ (used in the definition of whileLoop) using termination relations and invariants, instead of being forced to resort to induction.

Theorem 4.3 To show that a while-loop with condition C and body B terminates, it is sufficient to show that: (*i*) there exists a well-founded relation R on the loop's state/iterator; (*ii*) there also exists an invariant I that holds on the initial state/iterator of the loop; (*iii*) every loop iteration maintains the invariant I; and (*iv*) each loop iteration modifies the state/iterator pair such that the old and new pairs are in the relation R.

$$\frac{\begin{array}{cc}\mathsf{wf}\ R & I\ r\ s \\ \multicolumn{2}{c}{\forall r\ s.\ \{\!|\lambda s'.\ I\ r\ s' \wedge C\ r\ s' \wedge s' = s|\!\}\ B\ r\ \{\!|\lambda r'\ s'.\ I\ r'\ s' \wedge ((r', s'), (r, s)) \in R|\!\}}\end{array}}{\mathsf{whileLoop\_terminates}\ C\ B\ r\ s}$$

Proof Conceptually, a relation R is well-founded only if every path consisting solely of steps in R terminates after a finite number of such steps. We use Isabelle/HOL's in-built definition of wf, which allows us to carry out an induction proof on over the relation R when R is well-founded. As we know from our assumptions that each step of the loop both maintains the invariant I and obeys the relation R, every path through the loop must reach a final state in a finite number of steps. Hence, the loop will always terminate.

With the above termination rule in hand, we can now proceed to create a total correctness version of the Hoare rule Theorem 4.2:

Theorem 4.4 (WhileLoopNf) Given a while-loop with body B and condition C, then if (*i*) an invariant I holds on the initial state; (*ii*) at each loop iteration, the invariant continues to hold; (*iii*) at each loop iteration, the current iterator/state of the loop decreases with respect to a well-founded relation R; and (*iv*) while ever the invariant

holds, the loop's body will not fail; then the postcondition $(\lambda r\ s.\ I\ r\ s \wedge \neg\ C\ r\ s)$ will hold, and the loop as a whole will not fail:

$$\frac{\begin{array}{c}\mathsf{wf}\ R \\ \forall r_o\ s_o.\ \{\lambda s.\ I\ r_o\ s \wedge C\ r_o\ s \wedge s = s_o\}\ B\ r_o\ \{\lambda r\ s.\ I\ r\ s \wedge ((r,\ s),\ (r_o,\ s_o)) \in R\}\end{array}}{\{I\ r\}\ \mathsf{whileLoop}\ C\ B\ r\ \{\lambda r\ s.\ \neg\ C\ r\ s \wedge I\ r\ s\}}$$

Proof We need to prove three items: (*i*) the resulting states of the whileLoop satisfy $I\ r\ s$ and $\neg\ C\ r\ s$; (*ii*) the whileLoop does not fail because of non-termination; and (*iii*) the whileLoop does not fail because of failure of the body B.

We can prove the first item using Theorem 4.2, and the second with Theorem 4.3. The third item is shown by inducting over the whileLoop_results definitions; in particular, we show that—because the invariant I always holds after each step, and that the invariant I implies non-failure after each step—a failure state can never be reached. Thus, the loop as a whole also will not fail.

As we did with our previous while-loop rule, we rewrite this above rule into a format more convenient for automated reasoning by allowing the user to annotate the while-loop in advance with a custom invariant I and termination relation R using the whileLoop_inv constant:

$$\frac{\begin{array}{cc}\forall r_o\ s_o.\ \{\lambda s.\ I\ r_o\ s \wedge C\ r_o\ s \wedge s = s_o\}\ B\ r_o\ \{\lambda r\ s.\ I\ r\ s \wedge ((r,\ s),\ (r_o,\ s_o)) \in R\} & \\ \mathsf{wf}\ R & \forall r\ s.\ I\ r\ s \longrightarrow \neg\ C\ r\ s \longrightarrow Q\ r\ s\end{array}}{\{I\ r\}\ \mathsf{whileLoop\_inv}\ C\ B\ r\ I\ R\ \{Q\}}$$

Stronger rules for loop reasoning

The rules developed in the previous section allow us to carry out Hoare-style reasoning with our monadic while-loop combinator, but are still insufficient for general reasoning. Two problems remain:

- While the rules allow us to show *non-failure* of a particular loop, we can not use them in their current form to prove that failure actually *will* occur; and
- While the rules provide an upper bound on the possible states resulting from a particular loop, they don't provide a lower bound—that is, the rules allow us to state what results *can't* be produced, but not what states *will* be produced.

The second problem is a result of non-determinism: a loop might have one result, zero results, or an infinite number of results. The rules we have developed so far declare that if a result is produced, then it obeys an invariant I, but are not able to make any claim about the *number* of results.

These two constraints may not seem particularly limiting at first, but become a problem when we start wanting to carry out equational reasoning, such as performing

program simplifications. For example, without the ability to reason about the presence of loop failure, we cannot simplify infinite loops into monadic failure, like so:[7]

$$\mathsf{whileLoop}\ (\lambda\_\ \_.\ \mathsf{True})\ (\lambda\_.\ \mathsf{skip})\ r = \mathsf{fail}$$

In fact, any non-trivial equality statement involving the monadic while-combinator requires us to reason about precisely the set of results that could be returned. For example, the following function mult_by_add calculates the product $m \times n$ using only addition:

$$\begin{array}{l}\mathsf{mult\_by\_add}\ m\ n \equiv \\ \quad \textbf{do}\ (m,\ r) \leftarrow \mathsf{whileLoop}\ (\lambda(m,\ r)\ s.\ 0 < m) \\ \qquad\qquad\qquad\qquad (\lambda(m,\ r).\ \mathsf{return}\ (m - 1,\ r + n))\ (m,\ 0); \\ \qquad \mathsf{return}\ r \\ \quad \textbf{od}\end{array}$$

Using the rules already defined, we can prove the Hoare triple:

$$\{\lambda s.\ \mathsf{True}\}\ \mathsf{mult\_by\_add}\ m\ n\ \{\lambda rv\ s.\ rv = m * n\}$$

but would be unable to prove the stronger rule:

$$\mathsf{mult\_by\_add}\ m\ n = \mathsf{return}\ (m * n)$$

This latter rule, if we could prove it, would allow us to avoid having to reason about the function mult_by_add at all, but instead let us use Isabelle/HOL's simplifier to rewrite instances of the function call into a simple multiply operation.

We would like, then, to develop a stronger rule than those presented thus far, to enable us to carry out proofs such as those above without needing to constantly fall back to the base definitions of whileLoop. The following rule fulfils such a need, allowing us to show that an arbitrary loop of the form whileLoop $C\ B\ r_o\ s_o$ is equal to a particular value Q:

Theorem 4.5 (WhileLoopRule) A loop whileLoop $C\ B\ r_o\ s_o$ is equal to an arbitrary value Q if: (*i*) all states that can arise from the loop are in results Q; (*ii*) all states in results Q can be produced by the loop; (*iii*) if failed Q, then the loop also fails; (*iv*) if $\neg$ failed Q, then the loop does not fail:

$$\frac{\begin{array}{c}\{\!|\lambda s.\ s = s_o|\!\}\ \mathsf{whileLoop}\ C\ B\ r_o\ \{\!|\lambda r\ s.\ (r,\ s) \in \mathsf{results}\ Q|\!\} \\ \forall r_Q\ s_Q.\ (r_Q,\ s_Q) \in \mathsf{results}\ Q \longrightarrow \{\!|\lambda s.\ s = s_o|\!\}\ \mathsf{whileLoop}\ C\ B\ r_o\ \exists\{\!|\lambda r\ s.\ r = r_Q \wedge s = s_Q|\!\} \\ \mathsf{failed}\ Q \longrightarrow \mathsf{failed}\ (\mathsf{whileLoop}\ C\ B\ r_o\ s_o) \\ \neg\ \mathsf{failed}\ Q \longrightarrow \{\lambda s.\ s = s_o\}\ \mathsf{whileLoop}\ C\ B\ r_o\ \{\lambda\_\ \_.\ \mathsf{True}\}\end{array}}{\mathsf{whileLoop}\ C\ B\ r_o\ s_o = Q}$$

Proof The result follows from unfolding the definitions of valid, $\mathsf{valid}_{\mathsf{NF}}$ and $\mathsf{valid}_{\mathsf{EX}}$.

[7] Such infinite loops may appear in C programs to indicate program errors, such as when an assertion failure occurs. Simplifying these to fail statements better captures the programmer's intent than just leaving the loop as is.

If we know in advance that the loop will not fail (i.e., $\neg$ failed Q), we can simplify the above rule to:

$$\frac{\begin{array}{c}\{\!|\,\lambda s.\ s = s_o\,|\!\}\ \text{whileLoop}\ C\ B\ r_o\ \{\!|\,\lambda r\ s.\ (r, s) \in \text{results}\ Q\,|\!\} \\ \forall r_Q\ s_Q.\ (r_Q, s_Q) \in \text{results}\ Q \longrightarrow \{\!|\,\lambda s.\ s = s_o\,|\!\}\ \text{whileLoop}\ C\ B\ r_o\ \exists\{\!|\,\lambda r\ s.\ r = r_Q \wedge s = s_Q\,|\!\} \\ \neg\ \text{failed}\ Q\end{array}}{\text{whileLoop}\ C\ B\ r_o\ s_o = Q}$$

which combines the first and fourth assumptions into a single goal. As discharging these two subgoals frequently requires similar reasoning, this can help reduce proof effort.

While WhileLoopRule may at first appear to be simply rearranging the equation whileLoop $C\ B\ r\ s = Q$ into a slightly different representation, the individual subgoals are easier to tackle. In particular, the first and last subgoals can be discharged using the WhileLoopWp and WhileLoopNf theorems already presented. The remaining two subgoals can be discharged using the two theorems below:

Theorem 4.6 (WhileLoopExs) If a loop whileLoop $C\ B\ r$ executes from a state satisfying precondition P, then there will exist at least one state from the loop satisfying the postcondition Q, if (i) there exists an invariant I satisfied by the states meeting the precondition; (ii) there exists a well-founded relation R; (iii) each time the loop body is executed, *at least* one execution satisfies the invariant and decreases R; and finally (iv) $I\ r\ s \wedge \neg\ C\ r\ s$ implies the postcondition $Q\ r\ s$:

$$\frac{\begin{array}{cc}\forall s.\ P\ s \longrightarrow I\ r\ s & \\ \forall r\ so.\ \{\!|\,\lambda s.\ I\ r\ s \wedge C\ r\ s \wedge s = so\,|\!\}\ B\ r\ \exists\{\!|\,\lambda r'\ s'.\ I\ r'\ s' \wedge ((r', s'), (r, so)) \in R\,|\!\} & \\ \text{wf}\ R & \forall r\ s.\ I\ r\ s \longrightarrow \neg\ C\ r\ s \longrightarrow Q\ r\ s\end{array}}{\{\!|\,P\,|\!\}\ \text{whileLoop}\ C\ B\ r\ \exists\{\!|\,Q\,|\!\}}$$

Proof At a high level, we show that the postcondition Q can be satisfied by proving that there exists a finite-length path through it. The invariant I defines the path to be taken, and by insisting that every step along the path is in the well-founded relation R, we can be certain the path is finite.

The proof takes place simply by unfolding the definition of valid_{EX} and valid, and then inducting over the relation R. We show that from the initial state, there is always at least one state satisfying the invariant I until the loop exits.

Theorem 4.7 (WhileLoopFail) A loop whileLoop $C\ B\ r\ s$ is guaranteed to fail if (i) the loop executes at least once; (ii) there exists an invariant I satisfied by the starting state of the loop; and (iii) each time the loop body executes, there exists an execution that either fails or continues to satisfy the loop invariant:

$$\frac{\begin{array}{cc}I\ r\ s & C\ r\ s \\ \forall r.\ \{\!|\,\lambda s.\ I\ r\ s \wedge C\ r\ s \wedge \neg\ \text{failed}\ (B\ r\ s)\,|\!\}\ B\ r\ \exists\{\!|\,\lambda r'\ s'.\ C\ r'\ s' \wedge I\ r'\ s'\,|\!\} & \end{array}}{\text{failed}\ (\text{whileLoop}\ C\ B\ r\ s)}$$

Proof The loop can fail for two reasons: either there exists an execution which fails, or there exists an execution which is non-terminating. In the first case, the invariant I

defines a path leading to the failing execution;[8] we must prove that there is an execution satisfying I until we reach a state where failed $(B\ r\ s)$ holds. In the second case, I defines an infinite path; we must show that there is always an execution through the loop's body that remains on the path.

The proof of the theorem falls out by unfolding the definition of whileLoop and inducting over the definition of whileLoop_terminates.

With these rules in place, we are now in a position where we can prove the examples given at the beginning of this section without too much effort:

Example 4.1 To prove that an infinite loop is equal to the monadic fail command:

$$\mathsf{whileLoop}\ (\lambda\_\ \_.\ \mathsf{True})\ (\lambda\_.\ \mathsf{skip})\ r = \mathsf{fail}$$

we use WhileLoopRule, which, after unfolding the definition of fail and simplifying, gives us the two proof obligations:

1. $\forall s.\ \{\!|\lambda s_o.\ s_o = s|\!\}\ \mathsf{whileLoop}\ (\lambda\_\ \_.\ \mathsf{True})\ (\lambda\_.\ \mathsf{skip})\ ()\ \{\!|\lambda\_\ \_.\ \mathsf{False}|\!\}$
2. $\forall s.\ \mathsf{failed}\ (\mathsf{whileLoop}\ (\lambda\_\ \_.\ \mathsf{True})\ (\lambda\_.\ \mathsf{skip})\ ()\ s)$

The first is discharged using WhileLoopWp using the invariant $(\lambda\_\ \_.\ \mathsf{True})$, while the second is discharged using WhileLoopFail, again with the invariant $(\lambda\_\ \_.\ \mathsf{True})$.

Example 4.2 We can prove that the mult_by_add function defined above equals the simpler expression:

$$\mathsf{mult\_by\_add}\ m\ n = \mathsf{return}\ (m * n)$$

Because we are not expecting either side of the equation to fail, we used the non-failure variant of the WhileLoopRule, resulting in the following subgoals after simplifying:

1. $\forall s_o.\ \{\lambda s.\ s = s_o\}$
   $\mathsf{whileLoop}\ (\lambda(m,\ r)\ s.\ 0 < m)$
   $(\lambda(m,\ r).\ \mathsf{return}\ (m - 1,\ r + n))\ (m,\ 0)$
   $\{\lambda(m,\ r)\ s.\ m = 0 \wedge r = m * n \wedge s = s_o\}$
2. $\forall s_o.\ \{\!|\lambda s.\ s = s_o|\!\}$
   $\mathsf{whileLoop}\ (\lambda(m,\ r)\ s.\ 0 < m)$
   $(\lambda(m,\ r).\ \mathsf{return}\ (m - 1,\ r + n))\ (m,\ 0)$
   $\exists\{\!|\lambda(m,\ r)\ s.\ m = 0 \wedge r = m * n \wedge s = s_o|\!\}$

These two subgoals are discharged using the rules WhileLoopNf and WhileLoopExs, respectively. Because the body of the loop is fully deterministic, the proofs are the same: each uses an invariant I of $(\lambda(m,\ r)\ s.\ s = s_o \wedge r + m_o * n_o = m_o * n_o)$, where x_o represents the initial value of x, and a termination relation ensuring that the value of the loop's variable m decreases. Once these rules are applied, the remaining subgoals are discharged using simple arithmetic.

[8] We don't need to prove that the failing execution occurs in a finite number of steps; if there was an infinite path to the failing execution, the loop would fail because of non-termination, hence proving failed (whileLoop $C\ B\ r\ s$).

Example 4.3 We can also use WhileLoopRule to prove properties about loops that have non-terminating paths but nevertheless still produce results. For instance, consider the following loop:

$$\begin{array}{l}\mathsf{whileLoop}\ (\lambda(\mathit{cont},\ n)\ s.\ \mathit{cont})\\ \quad(\lambda(\mathit{cont},\ n).\ \textbf{do}\ \mathit{cont} \leftarrow \mathsf{select}\ \{\mathsf{True},\ \mathsf{False}\};\\ \qquad\qquad\qquad \mathsf{return}\ (\mathit{cont},\ n+1)\\ \qquad\qquad\quad \textbf{od})\ (\mathsf{True},\ 0)\end{array}$$

The loop begins with the variable n, a natural number, set to zero. Each loop iteration we increase the value n, and non-deterministically update the boolean variable *cont*. If *cont* is true, we continue executing the loop for another iteration. Only when *cont* becomes false do we exit the loop, returning the current value of n.

The loop is capable of returning any positive value of n—the non-deterministic choice simply has to return true n times. The loop also contains a single non-terminating path, when the non-deterministic choice never elects to exit the loop. Thus, we would expect that the loop is equivalent to:

$$((\{\mathsf{False}\} \times \{n.\ 0 < n\}) \times \{s\},\ \mathsf{True})$$

That is, the set of results that consists of *cont* set to False, n as a positive natural, and the state s unchanged. Additionally, the failure flag is set.

We can use WhileLoopRule once again to prove this. Applying the rule and simplifying the result, we are left with three subgoals to prove: (*i*) that every result of the loop has *cont* set to false, n strictly positive and the state unchanged; (*ii*) that for every positive value n, there exists a path through the loop that reaches n with *cont* set to false and the state unchanged; and (*iii*) the loop fails.

We show the first condition by using WhileLoopWp with the invariant:

$$I = (\lambda(\mathit{cont},\ n)\ s.\ s = s_o \wedge (\neg\ \mathit{cont} \longrightarrow 0 < n))$$

The second goal is shown by using WhileLoopNf with the invariant:

$$I = (\lambda(\mathit{cont},\ n)\ s.\ n \leq n_t \wedge \mathit{cont} \neq (n = n_t) \wedge s = s_o)$$

and a termination measure of $(n_t - n)$, where n_t represents the target value of n that we are trying to prove exists.

Finally, the third goal is shown simply using WhileLoopFail with the invariant $I = (\lambda\_\ \_.\ \mathsf{True})$, as the loop has an obvious non-terminating path through it.

4.4 Converting Simpl to a monadic representation

Now that we have a suitable representation for modelling imperative programs in Isabelle/HOL, we can proceed to transform the deeply embedded Simpl language into this new shallowly embedded monadic representation, while also generating a proof showing our translation is correct. We name the output of this initial translation stage *L1*.

```
TRY                                          catch_E
    IF {| 'a ≤_s 'b |} THEN                      (do condition_E (λs. a_' s ≤_s b_' s)
        'ret_int :== 'b;;                                (do modify_E (λs. s(| ret_int_' := b_' s |));
        'exn_var :== Return;;                                modify_E (λs. s(| exn_var_' := Return |));
        THROW                                                throw_E ()
    ELSE                                                  od)
        SKIP                                             skip_E;
    FI;;                                             modify_E (λs. s(| ret_int_' := a_' s |));
    'ret_int :== 'a;;                                modify_E (λs. s(| exn_var_' := Return |));
    'exn_var :== Return;;                            throw_E ();
    THROW;;                                          guard_E (λs. s ∈ ∅)
    GUARD DontReach ∅                             od)
        SKIP                                     (λ_. skip_E)
CATCH
    SKIP
END
```

Figure 4.1: A simple `max` function represented in Simpl (left-hand side) and its corresponding L1 monadic representation (right-hand side).

The conversion process is conceptually easy: Simpl constructs are simply substituted with their monadic equivalents shown in Table 4.4. For example, the Simpl statement Basic f, which updates the state s to the value $f\ s$, is translated to $\text{modify}_E\ f$. Similarly, the Simpl statement Cond $c\ L\ R$ is translated to condition $(\lambda s.\ s \in c)\ L'\ R'$, where L' and R' are the appropriate translations of the Simpl statements L and R, respectively. In this second case, we also convert the Simpl condition $c :: 'state\ set$ to the predicate form $(\lambda s.\ s \in c) :: 'state \Rightarrow bool$, which we find more intuitive. Figure 4.1 shows an example C program '`max`' represented in Simpl, and its equivalent L1 monadic representation.

Because Simpl programs generated by Norrish's C-to-Isabelle parser keep all data (including local variables) exclusively in the program's state, our translated program doesn't use the monadic framework's bound variables, but sets them all to be Isabelle/HOL's *unit* element "()".[9]

Function calls

Perhaps the most fiddly part of the conversion from Simpl to L1 is handling function calls. Simpl's Call *dest_fn* statement transfers control flow from the current function to the beginning of function *dest_fn* without taking any further action. When modelling C programs, however, we also need to ensure that (*i*) local variables in the calling function's scope are preserved; (*ii*) the calling function is able to pass function parameters to *dest_fn*; and (*iii*) the calling function is able to receive a return value from the function *dest_fn*.

To handle these requirements, Norrish's C-to-Isabelle parser doesn't use the Simpl

[9] We will revisit this decision in Chapter 5, when we modify our monadic programs to use the monadic bound variables to keep track of local variables, instead of storing them in the program's state.

```
call setup dest_fn teardown return_xf ≡
    DYNCOM
        (λs. TRY
                BASIC setup;;
                CALL dest_fn
             CATCH
                BASIC (teardown s);;
                THROW
             END;;
             DYNCOM
                (λt. BASIC (teardown s);;
                     return_xf s t))
```

```
call_L1 setup dest_fn teardown return_xf ≡
    do s ← gets_E (λs. s);
       catch_E
          (do modify_E setup;
              dest_fn
           od)
          (λ_. fail_E);
       t ← gets_E (λs. s);
       modify_E (teardown s);
       modify_E (return_xf t)
    od
```

Figure 4.2: The definition of Norrish's call wrapper (*left*), and our call_{L1} monadic equivalent (*right*).

Call statement directly, but rather the following wrapper around it. The definition is shown in Figure 4.2.

The call wrapper performs the following steps:

1. The DynCom statement saves a copy of the currently executing function's state into the bound variable *s*;
2. The function *setup* :: $'s \Rightarrow 's$ is called, which copies function parameters from local variables in the caller's scope to variables in *dest_fn*'s scope;
3. The destination function *dest_fn* is called;
4. The function *teardown* :: $'s \Rightarrow 's \Rightarrow 's$ merges the global state from the output of *dest_fn* with the saved local variable state *s* from the calling function; and finally
5. The function *return_xf* :: $'s \Rightarrow 's \Rightarrow 's$ copies the return value of *dest_fn* into a variable of the current function's scope.

The call wrapper additionally has support for handling exceptions that cross function boundaries, but this support is not required in our context of C verification.

Instead of attempting to expand out the rather complicated semantics of Norrish's call wrapper at this point in AutoCorres, we instead just create an analogous call_{L1} function. The definition is shown in Figure 4.2.

Each step in the call_{L1} statement carries out the monadic equivalent of its Simpl counterpart. Unlike Norrish's call wrapper, our call_{L1} statement fails if an exception crosses a function boundary.[10] This definition of call_{L1} simply models the complexity of Norrish's call wrapper, making no attempt to improve it. In Section 5.1.3, we look at how we can eliminate the complexity altogether.

[10] In practice, C programs modelled in Simpl will not raise exceptions that span function boundaries, so the difference is inconsequential.

4.4.1 Proving conversion

Our goal is not simply to carry out a conversion from Simpl to L1, but also to generate a proof showing that our conversion is sound.

The proof we generate is a *refinement* proof; that is, the input Simpl program is a refinement of the output monadic program (or alternatively, the output monadic program is an *abstraction* of the input Simpl program). Informally, a concrete program C is a refinement of an abstract program A if C exhibits a subset of the behaviours of A. Once we have shown refinement between A and C we can, for instance, prove that C exhibits particular behaviours simply by reasoning on the (hopefully) simpler abstract program A.

In order to prove refinement, we start by declaring a predicate $\text{corres}_{\text{L1}}$ stating that the two programs correspond to each other, defined as follows:

$$
\begin{array}{l}
\text{corres}_{\text{L1}}\ \Gamma\ A\ C \equiv \\
\quad \forall s.\ \neg\ \text{failed}\ (A\ s) \longrightarrow \\
\quad\quad (\forall t.\ \Gamma \vdash \langle C,\ \text{Normal}\ s\rangle \Rightarrow t \longrightarrow \\
\quad\quad\quad (\text{case}\ t\ \text{of} \\
\quad\quad\quad\quad \text{Normal}\ s' \Rightarrow (\text{Norm}\ (),\ s') \in \text{results}\ (A\ s) \\
\quad\quad\quad\quad \mid \text{Abrupt}\ s' \Rightarrow (\text{Exc}\ (),\ s') \in \text{results}\ (A\ s) \\
\quad\quad\quad\quad \mid \_ \Rightarrow \text{False})) \wedge \\
\quad\quad \Gamma \vdash C \downarrow \text{Normal}\ s
\end{array}
$$

The definition reads as follows: Given a Simpl context Γ mapping function names to function bodies, a monadic program A and a Simpl program C, then, assuming that the monadic program A does not fail: (*i*) for each normal execution of the Simpl program there is an equivalent normal execution of the monadic program; (*ii*) similarly, for each execution of the Simpl program that results in an exception, there is an equivalent monadic execution also raising an exception; and finally, (*iii*) every execution of the Simpl program terminates.

The final termination condition (i.e., that we guarantee the concrete program terminates) may initially seem surprising. Notice, however, that these conditions must only hold if A does not fail, while our definition of $\text{whileLoop}_{\text{E}}$ ensures that loops with non-terminating executions will raise the failure flag. Consequently, proving termination of C is reduced to proving non-failure of A.

The definition of $\text{corres}_{\text{L1}}$ is useful because it allows us to reason about our concrete Simpl program using the L1 representation. For example, we can prove Hoare triples on a Simpl program by lifting the reasoning to L1, as follows:

Theorem 4.8 Given a concrete Simpl program C and a corresponding L1 program A, then if we prove a Hoare triple on A, the result also holds on executions of the concrete program C as follows:

$$
\frac{\text{corres}_{\text{L1}}\ \Gamma\ A\ C \qquad \{P\}\ A\ \{\lambda rv.\ Q\},\{\lambda rv.\ E\} \qquad P\ s}{\Gamma \vdash \langle C,\ \text{Normal}\ s\rangle \Rightarrow \text{Normal}\ t \longrightarrow Q\ t}
$$

Proof Straightforward from the definitions of $\text{corres}_{\text{L1}}$ and valid_{NF}.

The key part of the above theorem is that we must prove non-failure of our abstract program A. Without proving non-failure, we have no guarantees that C is actually a refinement of A.

The $\text{corres}_{\text{L1}}$ predicate also allows us to prove termination and fault freedom of the concrete Simpl program by reasoning on the L1 program alone:

Theorem 4.9 Given a concrete Simpl program C and a corresponding L1 program A, if we can show that A does not fail, then the concrete program C also: (*i*) will not fault; (*ii*) will not get stuck; and (*iii*) will terminate:

$$\frac{\text{corres}_{\text{L1}}\ \Gamma\ A\ C \qquad \{\lambda\_.\ \text{True}\}\ A\ \{\lambda\_\ \_.\ \text{True}\}}{\Gamma \vdash \langle C,\ \text{Normal}\ s\rangle \not\Rightarrow \text{Fault}\ f} \qquad \frac{\text{corres}_{\text{L1}}\ \Gamma\ A\ C \qquad \{\lambda\_.\ \text{True}\}\ A\ \{\lambda\_\ \_.\ \text{True}\}}{\Gamma \vdash \langle C,\ \text{Normal}\ s\rangle \not\Rightarrow \text{Stuck}}$$

$$\frac{\text{corres}_{\text{L1}}\ \Gamma\ A\ C \qquad \{\lambda\_.\ \text{True}\}\ A\ \{\lambda\_\ \_.\ \text{True}\}}{\Gamma \vdash C \downarrow \text{Normal}\ s}$$

Proof Again, straightforward from the definitions of $\text{corres}_{\text{L1}}$ and valid_{NF}.

Proving correspondence between Simpl and L1

To actually prove correspondence between concrete Simpl programs and their monadic equivalents, we craft a set of rules that can be applied both compositionally and automatically. Leaf statements, such as Skip, Basic f or Spec r can be directly shown to correspond to their monadic counterparts. For example, we can use the following rule to show that a Simpl statement Basic f corresponds to an L1 statement modify f:

$$\text{corres}_{\text{L1}}\ \Gamma\ (\text{modify}_{\text{E}}\ m)\ (\text{Basic}\ m)$$

Compound statements, such as Simpl's Seq or Cond statements, simply require us to prove that the sub-terms of the statements also correspond. For example, the following rule states that the Simpl program Cond $c\ L'\ R'$ corresponds to the L1 program condition $(\lambda s.\ s \in c)\ L\ R$ if L' corresponds to L and R' corresponds to R:

$$\frac{\text{corres}_{\text{L1}}\ \Gamma\ L\ L' \qquad \text{corres}_{\text{L1}}\ \Gamma\ R\ R'}{\text{corres}_{\text{L1}}\ \Gamma\ (\text{condition}_{\text{E}}\ (\lambda s.\ s \in c)\ L\ R)\ (\text{Cond}\ c\ L'\ R')}$$

Assuming we ensure that our L1 abstraction is structurally equivalent to the Simpl input program, a proof of correspondence can be generated by simply applying a set of rules in a syntax-directed fashion. Table 4.5 shows the full list of rules for proving correspondence between Simpl programs and their L1 equivalents.

4.4.2 Function calls and recursion

Function calls present a minor difficulty when converting larger programs. Two difficulties arise: the first is that we cannot define a function f in Isabelle until the functions f calls have first been defined. The second difficulty is that we typically cannot prove a

$$\frac{}{\mathsf{corres}_{\mathsf{L}_1}\ \Gamma\ \mathsf{skip}_{\mathsf{E}}\ \mathsf{Skip}}\ \textsc{L1CorresSkip} \qquad \frac{}{\mathsf{corres}_{\mathsf{L}_1}\ \Gamma\ (\mathsf{modify}_{\mathsf{E}}\ m)\ (\mathsf{Basic}\ m)}\ \textsc{L1CorresModify}$$

$$\frac{}{\mathsf{corres}_{\mathsf{L}_1}\ \Gamma\ (\mathsf{throw}_{\mathsf{E}}\ ())\ \mathsf{Throw}}\ \textsc{L1CorresThrow} \qquad \frac{}{\mathsf{corres}_{\mathsf{L}_1}\ \Gamma\ (\mathsf{spec}_{\mathsf{E}}\ x)\ (\mathsf{Spec}\ x)}\ \textsc{L1CorresSpec}$$

$$\frac{\mathsf{corres}_{\mathsf{L}_1}\ \Gamma\ B\ B'}{\mathsf{corres}_{\mathsf{L}_1}\ \Gamma\ (\mathbf{do}\ \mathsf{guard}_{\mathsf{E}}\ (\lambda s.\ s \in G);\ B\ \mathbf{od})\ (\mathsf{Guard}\ f\ G\ B')}\ \textsc{L1CorresGuard}$$

$$\frac{\mathsf{corres}_{\mathsf{L}_1}\ \Gamma\ L\ L' \qquad \mathsf{corres}_{\mathsf{L}_1}\ \Gamma\ R\ R'}{\mathsf{corres}_{\mathsf{L}_1}\ \Gamma\ (L \gg\!=_{\mathsf{E}} (\lambda\_.\ R))\ (L';;\ R')}\ \textsc{L1CorresSeq}$$

$$\frac{\mathsf{corres}_{\mathsf{L}_1}\ \Gamma\ L\ L' \qquad \mathsf{corres}_{\mathsf{L}_1}\ \Gamma\ R\ R'}{\mathsf{corres}_{\mathsf{L}_1}\ \Gamma\ (\mathsf{catch}_{\mathsf{E}}\ L\ (\lambda\_.\ R))\ (\mathsf{Catch}\ L'\ R')}\ \textsc{L1CorresCatch}$$

$$\frac{\mathsf{corres}_{\mathsf{L}_1}\ \Gamma\ L\ L' \qquad \mathsf{corres}_{\mathsf{L}_1}\ \Gamma\ R\ R'}{\mathsf{corres}_{\mathsf{L}_1}\ \Gamma\ (\mathsf{condition}_{\mathsf{E}}\ (\lambda s.\ s \in c)\ L\ R)\ (\mathsf{Cond}\ c\ L'\ R')}\ \textsc{L1CorresCond}$$

$$\frac{\Gamma\ X' = \mathsf{Some}\ Z' \qquad \mathsf{corres}_{\mathsf{L}_1}\ \Gamma\ Z\ Z'}{\mathsf{corres}_{\mathsf{L}_1}\ \Gamma\ Z\ (\mathsf{Call}\ X')}\ \textsc{L1CorresCall}$$

$$\frac{\mathsf{corres}_{\mathsf{L}_1}\ \Gamma\ f\ (\mathsf{Call}\ f')}{\begin{array}{l}\mathsf{corres}_{\mathsf{L}_1}\ \Gamma\ (\mathsf{call}_{\mathsf{L}_1}\ \mathit{setup}\ f\ \mathit{teardown}\ \mathit{ret\_xf}) \\ \quad (\mathsf{call}\ \mathit{setup}\ f'\ \mathit{teardown}\ (\lambda\_\ t.\ \mathsf{Basic}\ (\mathit{ret\_xf}\ t)))\end{array}}\ \textsc{L1CorresCallWrap}$$

$$\frac{\mathsf{corres}_{\mathsf{L}_1}\ \Gamma\ B\ B'}{\mathsf{corres}_{\mathsf{L}_1}\ \Gamma\ (\mathsf{whileLoop}_{\mathsf{E}}\ (\lambda\_\ s.\ s \in c)\ (\lambda\_.\ B)\ ())\ (\mathsf{While}\ c\ B')}\ \textsc{L1CorresWhile}$$

$$\frac{}{\mathsf{corres}_{\mathsf{L}_1}\ \Gamma\ \mathsf{fail}_{\mathsf{E}}\ X}\ \textsc{L1CorresFail}$$

Table 4.5: Selection of rules, compositionally proving $\mathsf{corres}_{\mathsf{L}_1}$ in the Simpl to L1 translation.

property about a function f until we know that the same property also holds for each function f calls.

For example, when proving the predicate $\text{corres}_{\text{L1}}$, function calls from f to g can only be proved by first showing that g is itself a valid abstraction of the concrete version of the function being called:

$$\frac{\Gamma\ X' = \text{Some}\ Z' \qquad \text{corres}_{\text{L1}}\ \Gamma\ Z\ Z'}{\text{corres}_{\text{L1}}\ \Gamma\ Z\ (\text{Call}\ X')}$$

For simple C programs, neither the problem of how to define functions or how to prove properties about functions are particularly difficult to solve: for both problems, we can simply topologically sort the entire program's callgraph and define or prove functions one by one, ensuring that every function is translated before its callers. By the time we reach the calling function, we will have the appropriate definition or proof about the callee, allowing the proof of the current function to succeed.

This simple approach of translating callees before their callers falls apart when we consider functions with recursion—especially functions that are mutually recursive. For instance, if function f calls function g, and function g calls function f, which should we translate first?

Even ignoring the difficulty of what order we should carry out our proofs or definitions in, recursive functions also reintroduce a problem we hit in Section 4.3 when determining how loops should be modelled in our monadic language: Isabelle is a logic of total functions, and will not let us define recursive functions until we first prove that they terminate. Proving termination is an undecidable problem in the general case and—even if we *could* automatically prove termination—we would still like to model C functions that don't necessarily terminate. A program that recursively iterates over a linked list is one such example: if the list terminates, so does the recursion; but if the list loops back on itself, then the recursion will continue forever.

Two potential solutions to defining and reasoning about recursive functions are as follows:

- One approach would be to model recursive functions using Krauss' *function* package [60], which provides mechanisms to allow potentially non-terminating recursive functions to be described and partially defined. The definitions of such functions are *conditional definitions*; that is, every function has a *domain* consisting of the set of terminating inputs. A function's definition for a particular input may be expanded only if the user first proves that the input is in the function's domain.

  Having such a conditional function definition would complicate automated reasoning on the function. In particular, our tool AutoCorres needs to carry out automated simplifications, induction and refinement proofs on generated functions. Having only a conditional functional definition would be a significant burden on engineering such proof procedures.

- Alternatively, we could give up on the idea of modelling recursive C functions using recursion in Isabelle, and convert recursive functions into non-recursive functions by explicitly rewriting each function to use a loop (using the monadic while-loop combinator) and an additional stack.

This approach has the advantage that AutoCorres only has to deal with recursion once; after that, the program just consists of standard loops. This comes at the cost, however, that the control flow of the function will be extensively contorted: local variables of a function must be saved to the faux stack in one iteration of the loop and restored from it later. Reasoning about the function would require extensive reasoning about the introduced stack, significantly complicating any proof. This problem gets worse when we consider mutual recursion, as we would have to combine multiple mutually recursive functions into a single loop body for the loop/stack conversion to take place; reasoning about the result would be unpleasant at best.

Neither of these solutions is particularly satisfactory. We instead introduce a simpler solution, described below.

Defining recursive functions

Our solution to representing recursive C functions in Isabelle is to introduce an additional parameter to every recursive function which we call a *measure*. When the function is initially called, the measure parameter is set to a natural m'. Each time the function recursively calls itself, it decrements the measure; if the measure reaches zero, the call to the function is defined as fail. So long as the initial measure value m' is larger than the depth of recursion, the function will return with the correct value.

Using such a measure parameter is a common technique in other interactive theorem provers such as ACL2 (where it is known as a *clock*), and in the context of Isabelle/HOL has the advantage that Krauss' function package is easily able to prove termination of the function: each recursive call simply decrements the measure parameter until it reaches zero. Because the function package proves termination for all inputs, we get an unconditional definition of the function, thus simplifying proofs.

Krauss' function package supports defining mutually recursive functions by providing the definitions of all the functions simultaneously. This allows AutoCorres to handle mutually recursive functions similarly to simple recursive functions: each call made from one of the mutually recursive functions to another decreases the measure. Like the simple recursion case, the function package is able to easily prove termination for the set of mutually recursive functions. Figure 4.3 gives an example of a pair of mutually recursive functions in C, and their hand-written monadic representation in Isabelle/HOL using this technique.

To simplify the implementation of AutoCorres we add such a measure parameter m' to *every* function translated, even if the function doesn't recurse. This avoids AutoCorres internally needing to treat functions with measure parameters differently from those without.[11] The measure parameter will remain through the various stages of AutoCorres.

[11] Additionally, some functions that are recursive in the input Simpl may not be recursive in later phases of AutoCorres. This may occur, for instance, if AutoCorres proves that the recursive call cannot occur during function optimisations. Because of this, AutoCorres *already* has to handle non-recursive functions with a measure parameter.

```
int is_even(unsigned n)
{
    if (n == 0)
        return 1;
    return is_odd(n - 1);
}

int is_odd(unsigned n)
{
    if (n == 0)
        return 0;
    return is_even(n - 1);
}

int test_odd(void)
{
    return is_odd(42);
}
```

$$
\begin{aligned}
&\mathsf{is\_even}'\ m'\ n \equiv \\
&\quad \mathsf{condition}\ (\lambda\_.\ m' = 0) \\
&\qquad \mathsf{fail} \\
&\qquad (\mathsf{condition}\ (\lambda\_.\ n = 0) \\
&\qquad\quad (\mathsf{return}\ 1) \\
&\qquad\quad (\mathsf{is\_odd}'\ (m' - 1)\ (n -_{\mathsf{w}} 1)))
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{is\_odd}'\ m'\ n \equiv \\
&\quad \mathsf{condition}\ (\lambda\_.\ m' = 0) \\
&\qquad \mathsf{fail} \\
&\qquad (\mathsf{condition}\ (\lambda\_.\ n = 0) \\
&\qquad\quad (\mathsf{return}\ 0) \\
&\qquad\quad (\mathsf{is\_even}'\ (m' - 1)\ (n -_{\mathsf{w}} 1)))
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{test\_odd}' \equiv \\
&\quad \mathsf{is\_odd}'\ (2^{32} + 1)\ 0\mathrm{x}2\mathrm{A}
\end{aligned}
$$

Figure 4.3: A pair of mutually recursive functions and a hand-written monadic definition of them in Isabelle. A measure parameter m' is initially set to $2^{32} + 1$ and decremented each time a recursive call takes place, allowing termination to be proven easily.

Unnecessary measure parameters will be removed in the last phase of AutoCorres, described in Section 5.4.

Automated proofs about recursive functions

The measure parameter not only allows termination to be automatically proven, but also has the benefit of simplifying automated proofs. In particular, the decreasing measure parameter allows AutoCorres to prove properties on recursive functions by inducting over the measure parameter of the functions. To prove an arbitrary property on such a definition, is suffices to show:

- The property is true when the measure m' equals zero. This corresponds to the monadic program fail, which trivially satisfies most properties of interest to us.
- Assuming that the property is true for $(m' - 1)$, AutoCorres must next prove that it remains true for m'. As our properties of interest tend to be compositional (that is, a rule is true for a function f if it is true for the body of f, and all functions that f calls), the recursive calls become no harder to reason about than a simple call to an unrelated function.

For example, to prove the correspondence theorem $\mathsf{corres}_{\mathrm{L1}}$ for the functions `is_even`, `is_odd` and `is_42_odd` shown in Figure 4.3, we would prove the following theorems:

$$\frac{\mathsf{corres}_{\mathsf{L}_1}\ \Gamma\ (\mathsf{is\_even}_{\mathsf{L}_1}\ 0)\ \mathsf{is\_even}_{\mathsf{C}} \quad \mathsf{corres}_{\mathsf{L}_1}\ \Gamma\ (\mathsf{is\_even}_{\mathsf{L}_1}\ (m' - 1))\ \mathsf{is\_even}_{\mathsf{C}} \quad \mathsf{corres}_{\mathsf{L}_1}\ \Gamma\ (\mathsf{is\_odd}_{\mathsf{L}_1}\ (m' - 1))\ \mathsf{is\_odd}_{\mathsf{C}}}{\mathsf{corres}_{\mathsf{L}_1}\ \Gamma\ (\mathsf{is\_even}_{\mathsf{L}_1}\ m')\ \mathsf{is\_even}_{\mathsf{C}}}$$

$$\frac{\mathsf{corres}_{\mathsf{L}_1}\ \Gamma\ (\mathsf{is\_odd}_{\mathsf{L}_1}\ 0)\ \mathsf{is\_odd}_{\mathsf{C}} \quad \mathsf{corres}_{\mathsf{L}_1}\ \Gamma\ (\mathsf{is\_even}_{\mathsf{L}_1}\ (m' - 1))\ \mathsf{is\_even}_{\mathsf{C}} \quad \mathsf{corres}_{\mathsf{L}_1}\ \Gamma\ (\mathsf{is\_odd}_{\mathsf{L}_1}\ (m' - 1))\ \mathsf{is\_odd}_{\mathsf{C}}}{\mathsf{corres}_{\mathsf{L}_1}\ \Gamma\ (\mathsf{is\_odd}_{\mathsf{L}_1}\ m')\ \mathsf{is\_odd}_{\mathsf{C}}}$$

$$\frac{\forall m'.\ \mathsf{corres}_{\mathsf{L}_1}\ \Gamma\ (\mathsf{is\_even}_{\mathsf{L}_1}\ m')\ \mathsf{is\_even}_{\mathsf{C}} \quad \forall m'.\ \mathsf{corres}_{\mathsf{L}_1}\ \Gamma\ (\mathsf{is\_odd}_{\mathsf{L}_1}\ m')\ \mathsf{is\_odd}_{\mathsf{C}}}{\mathsf{corres}_{\mathsf{L}_1}\ \Gamma\ \mathsf{test\_odd}_{\mathsf{L}_1}\ \mathsf{test\_odd}_{\mathsf{C}}}$$

Here $\mathsf{is\_even}_{\mathsf{C}}$ represents the Simpl representation of the function `is_even`, while $\mathsf{is\_even}_{\mathsf{L}_1}$ represents the L1 version of the function.

The first two theorems are the recursion base cases; we assume that the measure m' is zero, and show that the correspondence property holds. Because $(\mathsf{is\_even}_{\mathsf{L}_1}\ 0)$ and $(\mathsf{is\_odd}_{\mathsf{L}_1}\ 0)$ are both equivalent to fail, we can use the rule L1CorresFail to show that the result holds.

The next two theorems are the inductive case; we assume that the property holds for $(m' - 1)$, and must then prove it for m'. As recursive calls use a measure of $(m' - 1)$, to prove these rules we can simply use the standard L1CorresCall rule to handle the recursive call.

The final rule proves correspondence for `is_42_odd`, which is just a standard C function. We assume that both `is_even` and `is_odd` correspond for any measure m' to prove the property.

Once all the intermediate theorems have been proven independently, AutoCorres can then stitch them together using simple induction over the natural, forming our final correspondence theorems:

$$\frac{\forall m'.\ \mathsf{corres}_{\mathsf{L}_1}\ \Gamma\ (\mathsf{is\_even}_{\mathsf{L}_1}\ m')\ \mathsf{is\_even}_{\mathsf{C}} \quad \forall m'.\ \mathsf{corres}_{\mathsf{L}_1}\ \Gamma\ (\mathsf{is\_odd}_{\mathsf{L}_1}\ m')\ \mathsf{is\_odd}_{\mathsf{C}}}{\mathsf{corres}_{\mathsf{L}_1}\ \Gamma\ \mathsf{test\_odd}_{\mathsf{L}_1}\ \mathsf{test\_odd}_{\mathsf{C}}}$$

Choice of measure values

One final detail needs to be discussed: when calling into a recursive function f, what initial measure value m' should AutoCorres use? We could just fix the value—perhaps $2^{32} + 1$, or some other 'sufficiently large' constant—and simply hope for the best.

We can, however, give the user a little more flexibility by allowing her to choose the measure herself. Instead of fixing a measure value, for every recursive function f, AutoCorres declares the existence of a fixed function f_m of type $'state \Rightarrow nat$. Each time a call to a recursive function takes place, the generated function will pass the current state into f_m and use the value returned from the function as the initial measure. The user can then, after AutoCorres has run, define the declared function f_m to return a suitable value. In our `is_even` example above, for example, a suitable definition for a measure function would be to use the input parameter n, as we know that the function will never recurse more than n times.

Recursive measures compared to well-founded relations

Our choice to use measure parameters for recursive functions was to allow AutoCorres to automatically carry out proofs on the Isabelle functions it defined. In particular, almost every proof AutoCorres internally carries out on a recursive function requires inductive reasoning, and the addition of a measure parameter simplifies the implementation of such proof methods.

An alternative method of defining recursive functions that allows us to automatically prove properties such as termination would be to use well-founded relations, similar to those used in our whileLoop proof rules. For example, a monadic representation of a recursive factorial function using well-founded relations would be as follows:

```
factorial n ≡
    do s_o ← gets (λs. s);
       condition (λs. n = 0)
           (return 1)
           (do guard (λs. ((n − 1, s), (n, s_o)) ∈ R_fact ∧ wf R_fact);
               ret ← factorial (n − 1);
               return (n ∗ ret)
            od)
    od
```

This function definition uses a constant $\mathsf{R}_{\mathsf{fact}}$ that is declared but not defined. The user would be later responsible for providing a definition for this constant, consisting of a well-founded relation that shows that the function terminates.

The function factorial itself is defined as follows:

1. When the function is first entered, a copy of the current state is bound to the variable s_o;
2. If the function is in its base case (i.e., it has been called with $n = 0$), then it simply returns 1 to the caller;
3. Otherwise, a recursive call needs to take place. The function has a guard that ensures that the state and parameters passed to the recursive factorial call are 'smaller' with respect to $\mathsf{R}_{\mathsf{fact}}$ than the input state and parameters. The guard also ensures that the relation $\mathsf{R}_{\mathsf{fact}}$ is well-founded; this allows us to assume this fact when reasoning about recursive calls, despite the fact that $\mathsf{R}_{\mathsf{fact}}$ will not be defined by the user until after AutoCorres has finished.

   If both of the conditions of the guard are satisfied, then the recursive call takes place. Otherwise, the function simply fails.
4. Finally, once the recursive call returns, the function performs its factorial calculation, and returns this result.

In this case of this function, a valid definition of $\mathsf{R}_{\mathsf{fact}}$ would be any relation such that

$$((n', s'), (n, s)) \in \mathsf{R}_{\mathsf{fact}} \iff n' < n$$

which simply declares that the parameter n to the function decreases.

Importantly, functions defined using well-founded relations in such a manner can automatically be proven to terminate. The proof proceeds as follows: if $\mathsf{R}_{\mathsf{fact}}$ is not

well-founded, the guard statement in the function body will always fail, and no recursive calls will ever take place. Otherwise, R_{fact} is well-founded, and the guard above recursive call ensures that each recursive invocation of the function is 'smaller' with respect to R_{fact}, and hence the function will eventually terminate.

Defining functions using well-founded relations offers two primary benefits over the use of measures:

1. There exist functions that always terminate, but for which no measure value will be large enough. For example, imagine a function that when first called non-deterministically selects a positive natural number n, and then subsequently calls itself recursively n times with a single parameter that simply decreases until it reaches zero. While such a function will always terminate, no matter what measure value is chosen for this function there will always be an execution where the non-deterministically chosen parameter is larger, and hence termination cannot be proven. In contrast, it is not hard to construct a well-founded relation that can be used to prove termination for such a function.
2. Frequently, well-founded relations are easier to specify than measure values. For example, a function with arguments that always decrease in lexicographical order is trivial to specify using a well-founded relation, but harder with a measure.

In the context of C verification, functions where measures are insufficiently powerful to prove termination are unlikely to occur in practice: such functions exploit infinite non-determinism, which is not typically present in C programs.[12] The second argument—that well-founded relations can be easier for end users to specify—we find to be a more compelling argument, and may motivate changes to future versions of our tool to prefer them over measures.

4.5 Structural simplifications of monadic programs

The Simpl output generated by the C-to-Isabelle parser is, by design, as literal a conversion of C as possible. This frequently leads to specifications that are more complex than strictly necessary. This complexity ranges from the trivial, such as unnecessary $skip_E$ commands (perhaps generated from a stray semicolon in the C code, left behind after the C preprocessor stripped away debugging statements); to deeper complexity, such as exceptional control flow being used to model abrupt termination, as in the case of `return` and `continue` statements.

In this section, we look at two approaches that we can use to simplify the L1 representation of our C programs: *peephole optimisations*, which carry out localised simplifications to L1 specifications, and *exception elimination*, which performs a slightly deeper analysis of individual functions in an attempt to reduce their use of exceptional control flow.

[12] We say 'typically' because one could imagine such infinite non-determinism to appear in a concrete C program that interacts with an abstractly-modelled hardware device, the latter of which may exhibit infinite non-determinism.

$$\frac{\text{no\_throw } A}{\text{catch}_\text{E}\ A\ E = A}\ \textsc{CatchNoThrow} \qquad \frac{\text{no\_return } A}{A \gg=_\text{E} B = A}\ \textsc{BindAlwaysThrow} \qquad \text{catch}_\text{E}\ (\text{throw}_\text{E}\ a)\ E = E\ a\ \textsc{CatchThrow}$$

$$\frac{\text{no\_throw } A}{\text{catch}_\text{E}\ (A \gg=_\text{E} B)\ C = \textbf{do}\ x \leftarrow A;\ \text{catch}_\text{E}\ (B\ x)\ C\ \textbf{od}}\ \textsc{BindNoThrow}$$

$$\text{catch}_\text{E}\ (\text{condition}_\text{E}\ c\ L\ R)\ E = \text{condition}_\text{E}\ c\ (\text{catch}_\text{E}\ L\ E)\ (\text{catch}_\text{E}\ R\ E)\ \textsc{CatchCond}$$

$$\text{catch}_\text{E}\ (\text{condition}_\text{E}\ C\ L\ R \gg=_\text{E} B)\ E = \text{condition}_\text{E}\ C\ (\text{catch}_\text{E}\ (L \gg=_\text{E} B)\ E)\ (\text{catch}_\text{E}\ (R \gg=_\text{E} B)\ E)\ \textsc{CatchCondBind}$$

Table 4.6: Rules used by AutoCorres in the exception elimination phase.

4.5.1 Peephole optimisations

As the L1 specification is a shallow embedding, we are able to use Isabelle's rewrite engine to apply a series of peephole optimisations consisting of 23 rewrite rules, removing significant amounts of unnecessary code from the L1 programs. The rules are shown in Figure 4.4.

The rules fall roughly into four categories:

- *Code normalisation* rules convert L1 programs into a canonical form, helping other rules in the set to match;
- *Dead code elimination* rules remove unreachable or unnecessary code. This includes stripping unnecessary skip_E statements; removing guard_E statements that are always true; removing code after throw_E statements; and pruning branches of condition statements where the condition is True or False.
- *Failure propagation* rules propagate fail statements higher up the code. If a code-path will inevitably reach a fail statement, the code leading up to the statement can be stripped away; and
- *Simple loop elimination* rules rewrite loops that will never execute, infinite loops, and loops with failing bodies into simpler representations.

Even though the rules are simple, they produce a significant reduction in the size of program representations. The results in Table 5.8 on page 102 show that these rules alone reduced average term sizes by 22% on a larger project.

4.5.2 Exception elimination

Statements in C that cause abrupt termination, such as `return`, `continue` or `break`, are modelled in Simpl with exceptions, as described in Section 3.3. While exceptions

Code normalisation

$$f \gg\!=_{\mathsf{E}} (\lambda\_.\ g) \gg\!=_{\mathsf{E}} (\lambda\_.\ h) = f \gg\!=_{\mathsf{E}} (\lambda\_.\ g \gg\!=_{\mathsf{E}} (\lambda\_.\ h))$$

$$\textbf{do}\ A;\ \mathsf{skip}_{\mathsf{E}}\ \textbf{od} = A \qquad \textbf{do}\ \mathsf{skip}_{\mathsf{E}};\ A\ \textbf{od} = A$$

$$\mathsf{guard}_{\mathsf{E}}\ (\lambda\_.\ \mathsf{False}) = \mathsf{fail}_{\mathsf{E}} \qquad \mathsf{guard}_{\mathsf{E}}\ (\lambda\_.\ \mathsf{True}) = \mathsf{skip}_{\mathsf{E}}$$

Dead code elimination

$$\mathsf{condition}_{\mathsf{E}}\ (\lambda\_.\ \mathsf{True})\ A\ B = A \qquad \mathsf{condition}_{\mathsf{E}}\ (\lambda\_.\ \mathsf{False})\ A\ B = B \qquad \mathsf{condition}_{\mathsf{E}}\ C\ A\ A = A$$

$$\textbf{do}\ \mathsf{throw}_{\mathsf{E}}\ ();\ X\ \textbf{od} = \mathsf{throw}_{\mathsf{E}}\ () \qquad \textbf{do}\ \mathsf{fail}_{\mathsf{E}};\ X\ \textbf{od} = \mathsf{fail}_{\mathsf{E}}$$

Failure propagation

$$\textbf{do}\ \mathsf{skip}_{\mathsf{E}};\ \mathsf{fail}_{\mathsf{E}}\ \textbf{od} = \mathsf{fail}_{\mathsf{E}} \qquad \textbf{do}\ \mathsf{guard}_{\mathsf{E}}\ G;\ \mathsf{fail}_{\mathsf{E}}\ \textbf{od} = \mathsf{fail}_{\mathsf{E}}$$

$$\textbf{do}\ \mathsf{modify}_{\mathsf{E}}\ M;\ \mathsf{fail}_{\mathsf{E}}\ \textbf{od} = \mathsf{fail}_{\mathsf{E}} \qquad \textbf{do}\ \mathsf{init}_{\mathsf{E}}\ I;\ \mathsf{fail}_{\mathsf{E}}\ \textbf{od} = \mathsf{fail}_{\mathsf{E}}$$

$$\textbf{do}\ \mathsf{spec}_{\mathsf{E}}\ S;\ \mathsf{fail}_{\mathsf{E}}\ \textbf{od} = \mathsf{fail}_{\mathsf{E}} \qquad \textbf{do}\ \mathsf{fail}_{\mathsf{E}};\ \mathsf{fail}_{\mathsf{E}}\ \textbf{od} = \mathsf{fail}_{\mathsf{E}}$$

$$\mathsf{catch}_{\mathsf{E}}\ \mathsf{fail}_{\mathsf{E}}\ (\lambda\_.\ A) = \mathsf{fail}_{\mathsf{E}}$$

$$\mathsf{condition}_{\mathsf{E}}\ C\ \mathsf{fail}_{\mathsf{E}}\ A = \textbf{do}\ \mathsf{guard}_{\mathsf{E}}\ (\lambda s.\ \neg\ C\ s);\ A\ \textbf{od}$$

$$\mathsf{condition}_{\mathsf{E}}\ C\ A\ \mathsf{fail}_{\mathsf{E}} = \textbf{do}\ \mathsf{guard}_{\mathsf{E}}\ C;\ A\ \textbf{od}$$

$$\textbf{do}\ \mathsf{condition}_{\mathsf{E}}\ C\ L\ R;\ \mathsf{fail}_{\mathsf{E}}\ \textbf{od} = \mathsf{condition}_{\mathsf{E}}\ C\ (\textbf{do}\ L;\ \mathsf{fail}_{\mathsf{E}}\ \textbf{od})\ (\textbf{do}\ R;\ \mathsf{fail}_{\mathsf{E}}\ \textbf{od})$$

Simple loop elimination

$$\mathsf{whileLoop}_{\mathsf{E}}\ (\lambda\_\ \_.\ \mathsf{False})\ (\lambda\_.\ B)\ () = \mathsf{skip}_{\mathsf{E}}$$

$$\mathsf{whileLoop}_{\mathsf{E}}\ (\lambda\_.\ C)\ (\lambda\_.\ \mathsf{fail}_{\mathsf{E}})\ () = \mathsf{guard}_{\mathsf{E}}\ (\lambda s.\ \neg\ C\ s)$$

$$\mathsf{whileLoop}_{\mathsf{E}}\ (\lambda\_.\ C)\ (\lambda\_.\ \mathsf{skip}_{\mathsf{E}})\ () = \mathsf{guard}_{\mathsf{E}}\ (\lambda s.\ \neg\ C\ s)$$

Figure 4.4: The set of L1 peephole optimisation rules.

accurately model the behaviour of abrupt termination, their presence complicates reasoning about the final program: each block of code now has *two* exit paths that must be considered when reasoning about the program.

Fortunately, most function bodies can be rewritten to avoid the use of exceptional control flow by analysing where exceptions are actually used. For our analysis, we require two predicates to be defined. The first, no_throw, indicates that a given monadic code block will never raise an exception. The second no_return, indicates that a given monadic code block will never return normally. The predicates are defined as follows:

$$\begin{aligned}&\mathsf{no\_throw}\ f \equiv \\ &\qquad \{\!|\lambda s.\ \mathsf{True}|\!\}\ f\ \{\!|\lambda rv\ s.\ \mathsf{True}|\!\},\ \{\!|\lambda rv\ s.\ \mathsf{False}|\!\}\end{aligned}$$

$$\begin{aligned}&\mathsf{no\_return}\ f \equiv \\ &\qquad \{\!|\lambda s.\ \mathsf{True}|\!\}\ f\ \{\!|\lambda rv\ s.\ \mathsf{False}|\!\},\ \{\!|\lambda rv\ s.\ \mathsf{True}|\!\}\end{aligned}$$

For example, the following code block:

$$\begin{aligned}&\textbf{do}\ \mathsf{condition}_\mathsf{E}\ c \\ &\qquad (\mathsf{throw}_\mathsf{E}\ e_1) \\ &\qquad (\mathsf{modify}_\mathsf{E}\ f); \\ &\quad \mathsf{throw}_\mathsf{E}\ e_2 \\ &\textbf{od}\end{aligned}$$

satisfies the predicate no_return, because it always results in an exception being thrown. The fail_E statement satisfies both the no_throw statement and the no_return statement, as it neither raises an exception, nor does it ever return. These two predicates can be automatically proved on code blocks using a set of syntax-directed rules, listed in Table 4.7.

With the ability to determine the exception behaviour of blocks of code, we can now rewrite monadic code blocks to reduce their use of exceptional code flow; Table 4.6 shows the set of rewrite rules we use. For example, CatchNoThrow eliminates exception handlers surrounding code that never raises exceptions. Analogously, BindAlwaysThrow removes code trailing a block that *always* raises an exception.

Not all rules in this set can be applied blindly. In particular, the rules CatchCond and CatchCondBind duplicate blocks of code, which may trigger exponential growth in the size of our output programs in pathological cases. For CatchCond, which duplicates the exception handler, knowledge of our problem domain saves us: inputs originating from C only have trivial exception handlers automatically generated by Norrish's C-to-Isabelle parser, and hence duplicating them is of no concern.

The rule CatchCondBind, however, also duplicates its tail B, which may be arbitrarily large. While we can soundly apply the rule at any opportunity, we use the following heuristics to determine when applying the rule is likely to result in a simpler output program: (*i*) if neither branch of the condition throws an exception, then the rule BindNoThrow is applied; (*ii*) if both branches throw an exception, then BindAlwaysThrow is applied; (*iii*) if one branch always throws an exception, then the rule CatchCondBind is applied followed by BindAlwaysThrow on that branch, resulting in only a single instance of B in the output; finally (*iv*) if the body B is trivial, such as a $\mathsf{return}_\mathsf{E}$ or $\mathsf{throw}_\mathsf{E}$ statement, we apply CatchCondBind and duplicate B under the assumption that the

No throw rules

$$\mathsf{no\_throw}\ \mathsf{skip}_E \qquad \mathsf{no\_throw}\ (\mathsf{modify}_E\ m) \qquad \mathsf{no\_throw}\ (\mathsf{guard}_E\ g)$$

$$\mathsf{no\_throw}\ \mathsf{fail}_E \qquad \mathsf{no\_throw}\ (\mathsf{init}_E\ a) \qquad \mathsf{no\_throw}\ (\mathsf{spec}_E\ a)$$

$$\frac{\mathsf{no\_throw}\ L \quad \mathsf{no\_throw}\ R}{\mathsf{no\_throw}\ (\mathbf{do}\ L;\ R\ \mathbf{od})} \qquad \frac{\mathsf{no\_throw}\ L \quad \mathsf{no\_throw}\ R}{\mathsf{no\_throw}\ (\mathsf{condition}_E\ C\ L\ R)}$$

$$\frac{\mathsf{no\_throw}\ B}{\mathsf{no\_throw}\ (\mathsf{whileLoop}_E\ (\lambda\_.\ C)\ (\lambda\_.\ B)\ ())}$$

$$\frac{\mathsf{no\_throw}\ L}{\mathsf{no\_throw}\ (\mathsf{catch}_E\ L\ (\lambda\_.\ R))} \qquad \frac{\mathsf{no\_throw}\ R}{\mathsf{no\_throw}\ (\mathsf{catch}_E\ L\ (\lambda\_.\ R))}$$

No return rules

$$\mathsf{no\_return}\ (\mathsf{throw}_E\ ()) \qquad \mathsf{no\_return}\ \mathsf{fail}_E$$

$$\frac{\mathsf{no\_return}\ L}{\mathsf{no\_return}\ (\mathbf{do}\ L;\ R\ \mathbf{od})} \qquad \frac{\mathsf{no\_return}\ R}{\mathsf{no\_return}\ (\mathbf{do}\ L;\ R\ \mathbf{od})}$$

$$\frac{\mathsf{no\_return}\ L \quad \mathsf{no\_return}\ R}{\mathsf{no\_return}\ (\mathsf{catch}_E\ L\ (\lambda\_.\ R))} \qquad \frac{\mathsf{no\_return}\ L \quad \mathsf{no\_return}\ R}{\mathsf{no\_return}\ (\mathsf{condition}_E\ C\ L\ R)}$$

Table 4.7: Rules for proving the predicates no_throw and no_return on monadic program fragments.

Before exception elimination

```
catch_E
    (do condition_E (λs. a_' s ≤_s b_' s)
            (do modify_E (λs. s(|ret_int_' := b_' s|));
                modify_E (λs. s(|exn_var_' := Return|));
                throw_E ()
             od)
            skip_E;
        modify_E (λs. s(|ret_int_' := a_' s|));
        modify_E (λs. s(|exn_var_' := Return|));
        throw_E ();
        guard_E (λs. s ∈ ∅)
     od)
    (λ_. skip_E)
```

After exception elimination

```
do init_E ret_int_'_update;
    condition_E (λs. a_' s ≤_s b_' s)
        (do modify_E (λs. s(|ret_int_' := b_' s|));
            modify_E (λs. s(|exn_var_' := Return|))
         od)
        (do modify_E (λs. s(|ret_int_' := a_' s|));
            modify_E (λs. s(|exn_var_' := Return|))
         od)
od
```

Figure 4.5: An implementation of the `max` function converted to L1 before and after exception elimination takes place.

rewritten specification will still be simpler than the original. Otherwise, we leave the specification unchanged and let the user reason about the exception rather than generate a larger output specification.

Using these rules, all exceptions can be eliminated other than those in nested condition blocks described above, or those caused by `break` or `return` statements inside loop bodies.[13] Figure 4.5 shows the C `max` function (shown previously in Figure 4.1) before and after exception elimination optimisations have been applied. In this case, all exceptional control flow could be eliminated.

More generally, applying this transformation to the seL4 microkernel [57], 96% of functions could be rewritten to eliminate exceptional control flow. Of the remaining 4%, 10 could not be rewritten due to nested condition blocks, 13 because of either `return` or `break` statements inside a loop, and one function for both reasons independently.

[13] You could imagine rewriting such programs to avoid exceptional control flow by introducing an auxiliary variable indicating if the loop should be exited at the next iteration. Once you add in appropriate abstractions for reasoning about this variable, you will find that you have simply re-invented the exception monad, and not actually simplified the program.

4.6 Related work

Both Myreen et al. [76–78] and Li [68] have investigated the problem of automatically and verifiably translating deeply embedded representations of assembly language into shallowly embedded representations. Myreen et al. translate assembly into a representation consisting of let blocks, while Li targets a monadic representation closer to our own. Both Myreen et al. and Li model loops using tail-recursive functions, where termination need not be proven. The initial translation steps in this thesis are similar to those in Myreen et al. and Li's work, but the longer-term goals of the two projects are quite different—our program-level peephole optimisations and control flow simplifications don't have a direct equivalent in the context of assembly verification.

In work that attempts to model imperative programs using a monadic representation, the most common approach of modelling loops is simply to use recursion. This approach was taken in Bulwahn et al.'s [21] formalisation of monads in the *Imperative HOL* project, for instance. Isabelle/HOL has a (non-monadic) *while combinator*, along with a variety of rules to assist reasoning about it. This while combinator, however, would be unwieldy to use to model programs that have both non-determinism and non-terminating paths. In particular, the while combinator is only defined for terminating computations; in contrast, our own while-loop combinator still correctly calculates the results of terminating paths, even if non-terminating paths are also present.

Lammich and Tuerk [64] published work in parallel to our own, also formalising a monadic while-loop. Their motivations are similar to our own, though we need to handle additional complexities (such as failure and non-determinism) that Lammich and Tuerk did not face. Lammich and Tuerk additionally require two monadic loop constructs—one for partial correctness and one for total correctness. In contrast, we use the failure flag of our monad to indicate non-termination, allowing the same while-loop combinator to be used in both scenarios.

4.7 Conclusion

In this chapter we have shown how we can both automatically and verifiably transform programs represented in Schirmer's Simpl language into a monadic representation. This new representation allows us to easily apply peephole optimisations which simplify the structure of the program. Such optimisations—while not impossible to carry out—would be harder to implement if we attempted to carry them out directly on the deeply embedded Simpl language.

Observant readers may have noticed that we have not yet actually *reasoned* about any real C program. This is primarily because our L1 representation of programs—while easier to work with than the Simpl representation—is still unsatisfactory. In the next chapter, we look at how we can carry out further transformations in order to produce a representation of our program that we are actually willing to work with.

Chapter Summary

- When reasoning about the semantics of imperative programs, a shallow embedding is typically a more convenient representation than a deep embedding. Norrish's C-to-Isabelle parser generates a deeply embedded representation of C, however.
- We use Cock et al.'s existing formalisation of monads to model imperative programs. The formalisation supports modelling global state, non-determinism, failure, and exceptions. It does not support modelling loops, however.
- We extended Cock et al.'s formalisation with a new while-loop combinator, which allows imperative loops to be modelled. We additionally developed reasoning rules to ease manual reasoning about the combinator.
- We developed rules that allow Simpl programs to be automatically translated into their equivalent monadic representation. Our automatic translation uses these rules to generate a proof in Isabelle/HOL that our translation is correct.
- Finally, we developed optimisations that apply to our generated monadic output. These include peephole optimisations, which simplify control flow, and exception elimination rules, which remove the use of exceptions from programs where possible.

5 Local variable lifting

In the previous chapter, we saw how we can convert the deeply embedded Simpl language into a shallowly embedded monadic representation named L1. The L1 representation has a few benefits: for instance, it allows us to easily carry out simplifications to the program structure, such as eliminating exceptional control flow in many circumstances and carrying out other peephole optimisations. For end-users, the L1 representation is still deeply unsatisfying, however. Even the most basic reasoning about local variables remains needlessly complex.

In this chapter, we look at how we can improve our monadic representation of programs. We translate the L1 representation into a new representation, named L2, where local variables are no longer modelled as part of the program's global state, but instead modelled as monadic bound variables. As in the previous chapter, our goal is to not only transform the program specification into local lifted form, but also generate a proof of correctness that our transformation is correct.

The rest of this chapter is organised as follows: the next section describes how AutoCorres generates and proves correctness of local lifted form from L1 inputs. Section 5.2 looks at simplifications we can carry out on L2 specifications, that help to reduce their size and complexity. Section 5.3 considers how AutoCorres can further simplify L2 programs by using a simpler monad (or no monad at all) when possible. Finally, Section 5.4 describes how the various transformations are brought together for the end-user of AutoCorres.

This chapter is based on the published work by Greenaway et al. [49], *Bridging the gap: automatic verified abstraction of C* in ITP 2012.

5.1 Lifting local variables out of the program's state

Like the original Simpl embedding, our L1 translation represents local variables as part of the state: each time a local variable is read it is extracted from the state, and each

```
do modify_E (a_'_update (λ_. 3));
   condition_E (λs. 5 ≤_w a_' s)
       (modify_E (b_'_update (λ_. 5)))
       (modify_E (c_'_update (λ_. 4)));
   modify_E (λs. s(| ret__int_' := a_' s |))
od
```

(a) Locals in state

```
do a ← return_E 3;
   (b, c) ← condition_E (λs. 5 ≤_w a)
                (return_E (5, c))
                (return_E (b, 4));
   return_E a
od
```

(b) Local lifted form

Figure 5.1: Two program listings. The first stores local variables in the state, while the second uses bound variables. The shaded region does not affect the final return value; this is obvious in the second representation, but requires analysis of the shaded region to prove in the first representation.

time a local variable is written the state is modified. While this representation is easy to generate, it complicates reasoning about variable usage. An example of this is shown in Figure 5.1(a): the variable *a* is set to the value 3 at the top of the function and then later returned unmodified at the end of the function. However, to prove that the function returns the value 3, the user must first prove that the shaded part of the program preserves *a*'s value.

A better approach to representing local variables is to use the bound variables feature provided by Cock et al.'s monadic framework that we have thus far ignored. To achieve this, we remove local variables from the state type and instead model them as bound Isabelle/HOL variables. We name this representation *lifted local form* and the output of this translation *L2*. The representation is analogous to *static single assignment* (SSA) form used by many compilers as an intermediate representation [73], where each variable is assigned precisely once.

Figure 5.1(b) shows the same program in lifted local form. The function returns the variable *a*, which is bound to the value 3 in the first line of the function. As variables cannot change once bound, the user can trivially determine that the function returns 3 without inspecting the shaded area.

Two complications arise in representing programs in local lifted form. The first is that variables bound inside the bodies of $\mathsf{condition}_E$ and catch_E blocks are not available to statements after the block. To overcome this, we modify the bodies of such blocks to return a tuple of all variables assigned in the bodies and subsequently referenced, as demonstrated in Figure 5.1(b). Statements following the block can then use the names returned in this tuple.

A second, similar complication arises from loops, where local variables bound in one iteration not only need to be accessible *after* the loop, but also accessible by statements in the *next iteration* of the loop. We solve this by passing all required local variables between successive iterations of the loop as well as the result of the loop in the iterator of the $\mathsf{whileLoop}_E$ combinator. The function `loop` in Figure 5.2 shows such an example, where variables `a` and `c` are required between loop iterations and after the loop, respectively, and hence are passed through the loop iterator.

The actual process we use to lift local variables from the program's state into monadic bound variables takes place in three stages:

```
unsigned loop(unsigned a,
       unsigned b, unsigned c)
{
    unsigned d;
    while (a < b) {
        d = 1;
        c = a + b + d;
        a += 1;
    }
    return c;
}
```

```
do (a, c) ←
    whileLoop (λ(a, c) s. a <w b)
        (λ(a, c).
            do d ← return 1;
               c ← return (a +w b +w d);
               a ← return (a +w 1);
               return (a, c)
            od) (a, c);
    return c
od
```

Figure 5.2: A `while` loop in C translated into lifted local form. Variables that are live between loop iterations are passed through the loop iterator; in this example, `c` and `a`.

1. The program is analysed to determine how local variables are used in the body of each function, such as which local variables need to be bound to monadic variables, and which local variables can simply be discarded;
2. A new L2 monadic specification is generated from the existing L1 monadic specification, using the information gathered from the previous step. The algorithm that carries out this step—while well tested—is unproven; so finally,
3. A formal proof is constructed showing that the existing L1 specification is a refinement of the newly generated L2 specification.

We cover each of these stages in turn below.

5.1.1 Analysing existing local variable usage

Our first step in generating the local lifted form of a program is to analyse the program to determine how local variables are used within the function. Functions are broken down into *blocks*, which are simply L1 monadic statements. For example, the program fragment

```
do modifyE (...);
   skipE;
   failE
od
```

contains five monadic blocks. This can be more readily seen if we strip away the **do**/**od** syntax, like so:

$$\underbrace{\underbrace{\mathsf{modify}_\mathsf{E}\ (\ldots)}_{1}\ \gg\!=_\mathsf{E}\ (\ \lambda\_.\ \underbrace{\underbrace{\mathsf{skip}_\mathsf{E}}_{2}\ \gg\!=_\mathsf{E}\ (\lambda\_.\ \underbrace{\mathsf{fail}_\mathsf{E}}_{3}\ )}_{4}\ )}_{5}$$

The leaf statements $\mathsf{modify}_\mathsf{E}$, skip_E, fail_E form a block each. The bind_E statement $\mathsf{skip}_\mathsf{E} \gg\!=_\mathsf{E} (\lambda\_.\ \mathsf{fail}_\mathsf{E})$ forms another block, while the combination of the $\mathsf{modify}_\mathsf{E}$ statement and the inner bind_E forms the fifth block.

```
do modify_E (λs. s(| a_' := 1 |));
   modify_E (λs. s(| b_' := 2 |));
   modify_E (λs. s(| c_' := a_' s +_s b_' s |));
   modify_E (λs. s(| d_' := a_' s |));
   modify_E (λs. s(| c_' := 4 |));
   modify_E (λs. s(| e_' := c_' s |))
od
```

Read: {a, b}, *Write:* {c}, *Live Entry:* {a, b}, *Live Exit:* {a}.

(a) Linear program control flow

```
do whileLoop_E (λ_ s. a_' s <_w b_' s)
       (λ_. do modify_E (λs. s(| d_' := 1 |));
               modify_E (λs. s(| c_' := a_' s +_w b_' s +_w d_' s |));
               modify_E (λs. s(| a_' := a_' s +_w 1 |))
          od) ();
   modify_E (λs. s(| ret__unsigned_' := c_' s |))
od
```

Read: {a, b, d}, *Write:* {a, c, d}, *Live Entry:* {a, b}, *Live Exit:* {a, b, c}.

(b) Non-linear program control flow

Figure 5.3: Two examples of local variable usage analysis. Each example shows the set of variables read, the set of variables written, the set of variables that are live entering into the block, and the set of variables that are live at the end of the block.

For each block, we calculate four pieces of information:

- The *write set* of the block. This is the set of local variables that are potentially written to by the block;
- The *read set* of the block. This is the set of local variables potentially read from inside the block;
- The *entry live set* of the block. This is the set of local variables that, at the beginning of the block, contain a value that will possibly be read from (either in the current block, or any future block); and finally,
- The *exit live set* of the block. This is the set of local variables that, at the end of the block, contain a value that will possibly be read from by another block.

Figure 5.3 gives two examples. In the first example, we consider the shaded block, a $\mathsf{modify}_\mathsf{E}$ statement. The block writes to the variable c and reads from the variables a and b; these form the block's write and read sets, respectively. Variables a and b are live entering into the block, but the only variable live at the end of the block is the variable a: the variable b is never read from again, so is no longer live. The variable c, although read again, is unconditionally written to first so, while c becomes live again in the future, it is not live at the end of the shaded block.

In the second example, we consider the body of a loop. The body reads from the variables a, b and d and writes to the variables a, c and d; these form the block's read

and write sets respectively. The block uses values of a and b that originate from outside the scope of the loop's body, so these form the loop's entry live set. Finally, the variables a, b and c are live exiting the loop's body: a and b will be required if the loop carries out another iteration (and also just to carry out the loop's condition test), while c is required if the loop happens to finish. Because we don't know which path will be taken, we consider all three variables live.

To calculate the write set, read set, and live sets for each program block, AutoCorres first parses the expressions inside each program statement, and then uses this information to carry out a standard live variable analysis algorithm used by compilers. The implementation of these algorithms is not proven correct; an incorrect analysis will either result in the correspondence proofs described below failing (if the sets are missing live variables), or generated abstractions being unnecessarily verbose (if the sets have extraneous variables).

5.1.2 Utilising monadic return values

In monadic program representations, each program block *returns* a value. In our L1 program representation, the return value of every block was simply Isabelle's *unit* type. In our L2 program representation, each program block will return local variables that are both modified and still live at the end of the block. These return values can then be bound to a monadic variable and used later in the function. For example, the L1 statement

$$\mathsf{modify}_\mathsf{E}\ (\lambda s.\ s(\!|\, \mathsf{a\_}' := \mathsf{b\_}'\ s +_\mathsf{s} 1 \,|\!))$$

updates the local variable a; this could be converted to an L2 statement that returns the variable a, like so:

$$\mathsf{return}_\mathsf{E}\ (b +_\mathsf{s} 1)$$

While it appears that the write to variable a has been lost in the translation, if we place the line in the context of a larger program, we see how the return value of the $\mathsf{return}_\mathsf{E}$ statement can be bound to a variable a, as so:[1]

$$\begin{array}{l}\textbf{do}\ a \leftarrow \mathsf{return}_\mathsf{E}\ (b +_\mathsf{s} 1);\\ \quad \ldots\\ \textbf{od}\end{array}$$

Finally, as a minor implementation detail, we observe that the monadic gets_E function is a generalisation of the $\mathsf{return}_\mathsf{E}$ function:

$$\mathsf{return}_\mathsf{E}\ a = \mathsf{gets}_\mathsf{E}\ (\lambda s.\ a)$$

To avoid having to write rules in AutoCorres for handling both the $\mathsf{return}_\mathsf{E}$ and gets_E monadic functions, AutoCorres internally only ever uses gets_E. During the final *polish* phase of AutoCorres described in Section 5.4, calls to gets_E that do not depend on the

[1] We could have chosen any name for the bound variable; we choose the name a to match the input C source to help the end-user understand how the L2 output corresponds to her input program.

state are converted back into $\mathsf{return}_\mathsf{E}$ calls. In the rest of this document we use the $\mathsf{return}_\mathsf{E}\ v$ notation in our examples, but only show formal rules for $\mathsf{gets}_\mathsf{E}\ (\lambda s.\ v)$, which is what is used in our actual implementation.

Coercing return values of monadic blocks

By default, the return value of a monadic block translated from L1 to L2 will simply be that of the last statement in the block. For example, a simple L2 translation of the L1 program fragment

$$\begin{array}{l}\textbf{do}\ \mathsf{modify}_\mathsf{E}\ (\lambda s.\ s(\!|\, \mathsf{a\_}' := 1 \,|\!));\\ \quad \mathsf{modify}_\mathsf{E}\ (\lambda s.\ s(\!|\, \mathsf{b\_}' := 2 \,|\!));\\ \quad \mathsf{modify}_\mathsf{E}\ (\lambda s.\ s(\!|\, \mathsf{c\_}' := 3 \,|\!))\\ \textbf{od}\end{array}$$

will become

$$\begin{array}{l}\textbf{do}\ a \leftarrow \mathsf{return}_\mathsf{E}\ 1;\\ \quad b \leftarrow \mathsf{return}_\mathsf{E}\ 2;\\ \quad \mathsf{return}_\mathsf{E}\ 3\\ \textbf{od}\end{array}$$

Here, the return value of the block is the local variable `c`.

While AutoCorres often has *some* flexibility as to what variables a larger monadic block may return, in many circumstances a precise set of variables is required. For example, when converting the L1 condition statement

$$\begin{array}{l}\mathsf{condition}_\mathsf{E}\ (\ldots)\\ \quad (\mathsf{modify}_\mathsf{E}\ (\lambda s.\ s(\!|\, \mathsf{a\_}' := 1 \,|\!)))\\ \quad (\mathsf{modify}_\mathsf{E}\ (\lambda s.\ s(\!|\, \mathsf{b\_}' := 1 \,|\!)))\end{array}$$

a translation of the true-branch of the condition will by default return a new value for the variable `a`, while the false-branch will return a new value for the variable `b`. If we naïvely attempt to merge these two translations back into a new $\mathsf{condition}_\mathsf{E}$ statement, then it would be unclear precisely what variable the $\mathsf{condition}_\mathsf{E}$ would be returning.[2]

Instead, while converting from L1 to L2, we must at certain times *coerce* the return value of monadic blocks to return a different set of variables. In the example above, we would coerce both branches to return a tuple containing both the variables `a` and `b`, as follows:

$$\begin{array}{l}\mathsf{condition}_\mathsf{E}\ (\ldots)\\ \quad (\textbf{do}\ a \leftarrow \mathsf{return}_\mathsf{E}\ 1;\\ \qquad \mathsf{return}_\mathsf{E}\ (a,\ b)\\ \quad \textbf{od})\\ \quad (\textbf{do}\ b \leftarrow \mathsf{return}_\mathsf{E}\ 2;\\ \qquad \mathsf{return}_\mathsf{E}\ (a,\ b)\\ \quad \textbf{od})\end{array}$$

Now the true-branch of the condition statement returns the new value of `a` and the existing value of `b`, while the false-branch returns the opposite. The block as a whole returns both `a` and `b`.

[2] Not to mention that if the types of `a` and `b` were different, such a statement wouldn't even be type correct.

```
do a ← do (a, b) ← condition_E (...)
                   (return_E ...)
                   (return_E ...);
          return_E a
       od;
   modify_E (λs. s(| G_' := a |))
od
```

(a) Precise coercion

```
do (a, b) ← condition_E (...)
            (return_E ...)
            (return_E ...);
   modify_E (λs. s(| G_' := a |))
od
```

(b) Liberal coercion

Figure 5.4: Precise versus liberal coercion. In example (a) the left-hand side of the outer bind is coerced into returning precisely the set of variables required by the right-hand side, {a}. In example (b), the left-hand side returns a superset of the required variables, {a, b}; this avoids the extraneous return_E statement.

Coercion of blocks takes two possible forms: *precise* coercion, where a precise set of variables must be returned; and *liberal* coercion, where *at least* the given set of variables must be returned.

Liberal coercion is useful in situations where a block requires some variables from the previous block, but is able to disregard unnecessary values. By being liberal in the set of values accepted, unnecessary return_E statements may be avoided. For example, in Figure 5.4 the right-hand side requires access to the variable a; by allowing a superset of the required variables to be returned, we can avoid emitting an extraneous return_E statement, resulting in a simpler output.

AutoCorres carries out all of these coercions automatically during the translation from L1 to L2. Liberal coercion is used wherever possible in an effort to reduce the output complexity of the L2 output, while precise coercion is used wherever a strict set of variables is required.

5.1.3 Generating an L2 specification

Once local variable usage has been analysed for the input L1 program, each program block in the input is converted to an equivalent L2 block. We start by describing how simple leaf statements such as modify_E are converted, and then move on to describe how compound statements such as condition_E are translated.

In this section, we use the informal notation $P \blacktriangleright Q$ to state that program P is translated into Q.

Translating simple statements

skip_E and fail_E Simple parameterless statements such as skip_E and fail_E that do not access local variables can be easily translated into an L2 output command. fail_E is converted unchanged, while skip_E is rewritten into its definition $\text{gets}_E\ (\lambda s.\ ())$ in order

to reduce the number of monadic primitives AutoCorres must handle in later phases. In both cases the statements return the *unit* type.

$$\begin{aligned} \mathsf{skip}_E &\blacktriangleright \mathsf{gets}_E\ (\lambda s.\ ()) \\ \mathsf{fail}_E &\blacktriangleright \mathsf{fail}_E \end{aligned}$$

***guard*$_E$** guard_E statements that read local variables from the program's state are rewritten to use bound variables instead. Additionally, accesses to fields inside the globals record are lifted to direct accesses of the state. For instance, a simple guard_E statement is converted as follows:

$$\begin{aligned} &\mathsf{guard}_E\ (\lambda s.\ \mathrm{G\_}'\ (\mathsf{globals}\ s) <_w \mathrm{a\_}'\ s) \\ &\quad\blacktriangleright\ \mathsf{guard}_E\ (\lambda s.\ \mathrm{G\_}'\ s <_w a) \end{aligned}$$

***init*$_E$** The init_E statement, which updates the value of a variable to a non-deterministically chosen value, is translated into a $\mathsf{unknown}_E$ statement, which non-deterministically selects a value to be returned, defined as follows:

$$\mathsf{unknown}_E \equiv \mathsf{selectE}\ \mathsf{UNIV}$$

For example, the L1 statement that initialises the local variable `a` is translated as follows:

$$\mathsf{init}_E\ \mathrm{a\_}'\mathrm{\_update} \blacktriangleright \mathsf{unknown}_E$$

The translated statement returns the new (non-deterministically chosen) value for the local variable.

***modify*$_E$** modify_E statements in the L1 specification may either update a local variable or the global state. In the first case, our L2 translation should return the new value of the local variable, while in the second case our L2 translation should continue to modify the state.

AutoCorres parses the statement to determine which type of update a particular L1 modify_E statement is carrying out. If it is only updating a local variable, AutoCorres translates the statement into a gets_E statement, where the new value of the local variable is returned. If the program's global state is being updated, AutoCorres instead emits a modify_E statement, stripping away the unnecessary call to globals_update.

For example, the following L1 statement, which updates the local variable `a`, is converted as follows:

$$\begin{aligned} &\mathsf{modify}_E\ (\lambda s.\ s(\!|\mathrm{a\_}' := \mathrm{a\_}'\ s +_s 1|\!)) \\ &\quad\blacktriangleright\ \mathsf{gets}_E\ (\lambda s.\ a +_s 1) \end{aligned}$$

In this example, the gets_E statement returns the new value of the variable `a`. In contrast, the following L1 statement, which writes to the global variable `G`, is converted as follows:

$$\begin{aligned} &\mathsf{modify}_E\ (\lambda s.\ \mathsf{globals\_update}\ (\lambda s'.\ s'(\!|\mathrm{G\_}' := \mathrm{a\_}'\ s|\!))\ s) \\ &\quad\blacktriangleright\ \mathsf{modify}_E\ (\lambda s.\ s(\!|\mathrm{G\_}' := a|\!)) \end{aligned}$$

In this example the L2 output continues to have a $\mathsf{modify}_{\mathsf{E}}$ statement that updates the global state.

***call*$_{L1}$** The $\mathsf{call}_{\mathsf{L1}}$ statement has quite complex semantics, inherited from the C-to-Isabelle parser in order to deal with saving and restoring the local variable state when crossing function boundaries. Recall from Section 4.4 that a statement of the form

$$\mathsf{call}_{\mathsf{L1}}\ \mathit{setup}\ \mathit{dest\_fn}\ \mathit{teardown}\ \mathit{return\_xf}$$

carries out the following steps: (*i*) a copy of the currently executing function's state is saved; (*ii*) the function $\mathit{setup} :: 's \Rightarrow 's$ is called, which copies function parameters from local variables in the caller's scope to variables in *dest_fn*'s scope; (*iii*) *dest_fn* is executed; (*iv*) the function $\mathit{teardown} :: 's \Rightarrow 's \Rightarrow 's$ merges the global state from the output of *dest_fn* with the saved local variable state from the calling function; and finally (*v*) the function $\mathit{return\_xf} :: 's \Rightarrow 's \Rightarrow 's$ copies the return value of *dest_fn* into a variable of the current function's scope.

Because almost all of the complexity of $\mathsf{call}_{\mathsf{L1}}$ stems from saving, restoring and merging local variable state, in our output L2 representation we can avoid jumping through so many hoops: calling the function *dest_fn* will not affect locally bound variables, so we don't need *setup* or *teardown*. Additionally, L2 functions use their monadic result as their return value, so we no longer need *return_xf* to extract return values from the state.

In fact, we could *almost* replace $\mathsf{call}_{\mathsf{L1}}$ statements with a direct monadic call to the destination function. For reasons we will describe shortly, we instead replace $\mathsf{call}_{\mathsf{L1}}$ statements with a $\mathsf{call}_{\mathsf{L2}}$ statement, defined as follows:

$$\mathsf{call}_{\mathsf{L2}}\ x \equiv \mathsf{catch}_{\mathsf{E}}\ x\ (\lambda\_.\ \mathsf{fail}_{\mathsf{E}})$$

The monadic statement $\mathsf{call}_{\mathsf{L2}}\ f$ starts by executing function f. If the function returns normally, $\mathsf{call}_{\mathsf{L2}}\ f$ will simply return the result of f. If the function raises an exception, however, $\mathsf{call}_{\mathsf{L2}}\ f$ will fail.

Having a special function to deal with exceptions at first appears unnecessary: the C-to-Isabelle parser only uses exceptions to model abrupt termination of C statements such as `return`, `continue` and `break`; such 'exceptions' will never be thrown across function boundaries, hence it would seem like dealing with such exceptions is unnecessary.

Our reason for catching these never-occurring exceptions in the $\mathsf{call}_{\mathsf{L2}}$ constant is that AutoCorres can locally reason that they don't occur, instead of having to analyse the entire function being called. The (strictly unnecessary) $\mathsf{call}_{\mathsf{L2}}$ predicate will eventually be optimised away during the type strengthening process described in Section 5.3, meaning that the end-user of AutoCorres will never actually see it.

For example, a C function call of the form:

```
int target_fn(int d, int e, int f);
r = target_fn(a, b, c);
```

is translated as follows:

$$\begin{aligned}&\mathsf{call}_{\mathsf{L1}}\ (\lambda s.\ s(\!|\, \mathsf{d\_}' := \mathsf{a\_}'\ s,\ \mathsf{e\_}' := \mathsf{b\_}'\ s,\ \mathsf{f\_}' := \mathsf{c\_}'\ s \,|\!))\\ &\quad (\mathsf{target\_fn}_{\mathsf{L1}}\ m')\ (\lambda s_o\ s.\ s_o(\!|\, \mathsf{globals} := \mathsf{globals}\ s \,|\!))\\ &\quad (\lambda s_o\ s.\ s_o(\!|\, \mathsf{r\_}' := \mathsf{ret\_\_int\_}'\ s \,|\!))\\ &\blacktriangleright\ \mathsf{call}_{\mathsf{L2}}\ (\mathsf{target\_fn}_{\mathsf{L2}}\ m'\ a\ b\ c)\end{aligned}$$

where target_fn$_{L1}$ and target_fn$_{L2}$ are the L1 and L2 translations of the function `target_fn` respectively, and m' is the recursive measure parameter described in Section 4.4.2.

Translating compound statements

To convert compound L1 statements to L2, AutoCorres uses a recursive process where child blocks of the L1 statement are first translated and joined together. When carrying out such translations, AutoCorres must ensure that child blocks return appropriate sets of variables.

Whenever a parent block recursively translates a child block, the parent specifies three pieces of information:

1. A set of *required return* variables R, that the child block must return. When translating the child block, the block will be coerced to ensure that this set of variables is returned;
2. A flag indicating if the child block may use liberal coercion; that is, whether the parent requires a precise set of variables to be returned, or if additional variables may also be returned; and finally
3. A set of *required thrown* variables E, which specifies which variables all exception-triggering throw$_E$ statements in the block must throw.

The required return and required thrown sets of variables are determined based on the previously calculated liveness and variable modification information. When starting translation of a function, the required return set R is simply the variable containing the return value of the function (such as ret__int_'), while the required thrown set E is the empty set.

In the following descriptions, we use the notation M_x to denote the set of variables modified by the block x, and L_x to denote the set of variables live entering block x. Additionally, the notation R_x and E_x denotes the set of required-return and required-thrown variables, respectively, when translating block x

A $\gg\!=_E$ ***B*** A bind$_E$ statement simply executes the left-hand side A followed by the right-hand side B. A is translated with $R_A = L_B \cap M_A$ (i.e., the left-hand side must return any variables that it modifies and remain live entering B), while $R_B = R$. As the right-hand block B can freely ignore any extraneous variables returned by A, AutoCorres allows liberal coercion when translating A.

Prior to starting the L2 conversion, AutoCorres rewrites bind$_E$ statements to be in right-associative form, using the standard monadic rule BindAssoc:

$$((f \gg\!=_E g) \gg\!=_E h) = (f \gg\!=_E (\lambda x.\ g\ x \gg\!=_E h))$$

Such bind statements, when in right-associative form, allow the inner-most statement to have direct access to all variables bound by previous statements. This reduces the number of additional coercions that need to take place, significantly simplifying the output L2 specification. Figure 5.5 shows an example of the difference this makes in practice. The input L1 specification is in left-associative form. The L2 output labelled (b) shows a direct

```
modifyE (λs. s(| a_' := 1 |)) >>=E
  (λ_. modifyE (λs. s(| b_' := 2 |))) >>=E
  (λ_. modifyE (λs. s(| c_' := a_' s +w b_' s |)))
```

(a) Input L1 specification

```
do
  (a, b) ← do
             a ← returnE 1;
             b ← returnE 2;
             returnE (a, b)
           od;
  returnE (a +s b)
od
```

(b) L2 from left-associative form

```
do
  a ← returnE 1;
  b ← returnE 2;
  returnE (a +s b)
od
```

(c) L2 from right-associative form

Figure 5.5: A simple L1 input and two possible corresponding L2 outputs. The left L2 output is a direct translation of the L1 input, while the second is the resulting translation after first converting the L1 input into right-associative form.

translation from left-associative form; a coercion needs to be inserted after the second return_E statement so that the final line is able to access the bound variable *a*. In contrast, the L2 output labelled (c) is generated after applying the rewrite rule BindAssoc to the input L1 specification; the coercion has been avoided.

condition$_E$ c A B The condition_E statement evaluates an expression and then executes one of two branches. The type of condition_E requires that the left-hand side and right-hand side return the same set of variables. Thus when translating the child blocks, $R_A = R_B = R \cap (M_A \cup M_B)$. That is, both blocks must return variables required by the parent block that are modified on *either* side of the condition. An example of this translation is shown in Figure 5.6.

```
conditionE (λs. a_' s <w b_' s)
  (modifyE (λs. s(| a_' := 1 |)))
  (modifyE (λs. s(| b_' := 1 |)))
```

▶

```
conditionE (λs. a <w b)
  (do a ← returnE 1;
      returnE (a, b)
   od)
  (do b ← returnE 2;
      returnE (a, b)
   od)
```

Figure 5.6: An example of converting a condition_E block from L1 to L2.

whileLoop$_E$ c B The whileLoop_E executes the body of the loop while ever the loop condition remains true.

The whileLoop_E combinator has a loop iterator that allows variables bound in the loop body to be passed into the next iteration of the loop or returned out of the loop. When converting an L1 loop into an L2 loop, we use $R_B = (R \cup L_B) \cap M_B$, which also becomes the set of variables placed into the loop iterator. That is, variables modified in

Table 5.1: Summary of the set of required-return and required-thrown variable parameters when recursively translating a program from L1 to L2.

Statement	R_A	R_B	E_A	E_B
$\mathsf{condition}_\mathsf{E}\ c\ A\ B$	$R \cap (M_A \cup M_B)$	$R \cap (M_A \cup M_B)$	E	E
$A \gg\!=_\mathsf{E} B$	$L_B \cap M_A$	R	E	E
$\mathsf{catch}_\mathsf{E}\ A\ B$	R	R	$L_B \cap M_A$	E
$\mathsf{whileLoop}_\mathsf{E}\ c\ B\ i$	—	$(R \cup L_B) \cap M_B$	—	E

the loop body that are either live between loop iterations or required by our parent block are recorded in the loop iterator. An example of this translation is shown in Figure 5.7.

```
whileLoop_E (λ_ s. a_' s <_w b_' s)                    whileLoop_E (λ(a, b) s. a <_w b)
    (λ_. do                                                (λ(a, b).
          modify_E (λs. s(| a_' := a_' s +_w 1 |));  ▶        do
          modify_E (λs. s(| b_' := b_' s −_w 1 |))             a ← return_E (a +_s 1);
        od) ()                                                 b ← return_E (b −_s 1);
                                                               return_E (a, b)
                                                             od) (a, b)
```

Figure 5.7: An example of converting a $\mathsf{whileLoop}_\mathsf{E}$ block from L1 to L2.

catch*$_E$ *A B The $\mathsf{catch}_\mathsf{E}$ and associated $\mathsf{throw}_\mathsf{E}$ statements allow execution to transfer from a point in the body of the $\mathsf{catch}_\mathsf{E}$ directly to the handler. Variables bound in the body of the $\mathsf{catch}_\mathsf{E}$ block do not remain in scope in the handler; instead, AutoCorres must ensure that all variables required by the handler are passed into the handler by modifying $\mathsf{throw}_\mathsf{E}$ statements inside the body to pass the variables into the handler.

The return value of the blocks A and B is $R_A = R_B = R$; that is, we simply return the set of variables required by our parent block. The set of variables that must be thrown in block A is $E_A = (L_B \cap M_A)$; that is, any variable live entering into the exception handler B that is modified by block A. An example of this translation is shown in Figure 5.8.

```
                                                       catch_E
catch_E                                                    (do
    (do                                                      a ← return_E (a +_s 1);
      modify_E (λs. s(| a_' := a_' s +_w 1 |));              ...;
      ...;                                                   throw_E a;
      throw_E ();                                  ▶         ...;
      ...;                                                   return_E (a, b +_s 1)
      modify_E (λs. s(| b_' := a_' s +_w 1 |))           od)
    od)                                                    (λa. do
    (λ_. modify_E (λs. s(| b_' := a_' s −_w 1 |)))               b ← return_E (a −_s 1);
                                                                 return_E (a, b)
                                                               od)
```

Figure 5.8: An example of converting a $\mathsf{catch}_\mathsf{E}$ block from L1 to L2.

Table 5.1 gives a summary of the calculated needed-returns and needed-throw values

used in the recursive process. Once every statement has been recursively translated using the process described above, AutoCorres will have a version of the input L1 specification with the local variables lifted out of the state into bound variables.

5.1.4 Proving correspondence between L1 and L2

In the previous section, we described how AutoCorres translates an input L1 specification into L2 form. The process described is well tested, having been applied to several large projects, such as those described in Section 8.2. Such testing, however, does not provide any formal guarantee of correctness.

For the soundness proof of the translation from L1 to L2 we use a refinement property $\text{corres}_{\text{L2}}$ defined as follows:

$$
\begin{array}{l}
\text{corres}_{\text{L2}}\ st\ rx\ ex\ P\ A\ C \equiv \\
\quad \forall s.\ P\ s \wedge \neg\ \text{failed}\ (A\ (st\ s)) \longrightarrow \\
\quad\quad (\forall (r,\ t) \in \text{results}\ (C\ s). \\
\quad\quad\quad \text{case}\ r\ \text{of} \\
\quad\quad\quad\quad \text{Exc}\ (\,) \Rightarrow (\text{Exc}\ (ex\ t),\ st\ t) \in \text{results}\ (A\ (st\ s)) \\
\quad\quad\quad\quad \mid \text{Norm}\ (\,) \Rightarrow (\text{Norm}\ (rx\ t),\ st\ t) \in \text{results}\ (A\ (st\ s))) \wedge \\
\quad\quad \neg\ \text{failed}\ (C\ s)
\end{array}
$$

The predicate has several parameters: *st* is a state translation function, converting the L1 state type to the L2 state type by stripping away local variable data; *P* is a precondition used to ensure that input bound variables in the L2 program match their L1 values; and *A* and *C* are the abstract L2 and concrete L1 programs, respectively. The values *rx* and *ex* are a *return extraction function* and an *exception extraction function*. They are required because the L2 monads return or throw values, while the corresponding L1 monads store these values in their state. The return extraction function *rx* extracts a value out of the L1 state to compare with the return value of the L2 monad, while *ex* is used to compare an exception's payload with the corresponding L1 state.

The $\text{corres}_{\text{L2}}$ definition can be read as: for all states matching the precondition *P*, assuming that *A* executing from state *st s* does not fail, then the following holds:

1. For each normal execution of *C* there is an equivalent execution of *A* whose return value will match the value extracted using *rx* from *C*'s state;
2. Similarly, every exceptional execution of *C* will have an equivalent execution of *A* with an exception value that matches the value extracted using *ex* from *C*'s state; and finally,
3. The execution of *C* will not fail.

The first two conditions ensure that executions in L2 match those of L1 with locals bound accordingly. The last condition allows us to later reduce non-failure of L1 programs to non-failure of L2 programs.

```
max_L1 m' ≡
    do init_E ret__int_'_update;
       condition_E (λs. a_' s ≤_s b_' s)
           (do modify_E (λs. s(| ret__int_' := b_' s |));
               modify_E (global_exn_var_'_update (λ_. Return))
            od)
           (do modify_E (λs. s(| ret__int_' := a_' s |));
               modify_E (global_exn_var_'_update (λ_. Return))
            od)
    od
```

(a) Function `max` at end of L1

```
max_L2 m' a b ≡
    do ret ← unknown_E;
       ret ← condition_E (λs. a ≤_s b)
                 (do ret ← gets_E (λs. b);
                     global_exn_var ← gets_E (λ_. Return);
                     gets_E (λs. ret)
                  od)
                 (do ret ← gets_E (λs. a);
                     global_exn_var ← gets_E (λ_. Return);
                     gets_E (λs. ret)
                  od);
       return_E ret
    od
```

(b) Function `max` translated to L2

Figure 5.9: The `max` function shown at the end of the L1 phase and at the end of the L2 phase, after optimisations have been carried out.

As a concrete example, Figure 5.9 shows our example `max` function after local variable lifting has taken place. The generated corres_{L2} predicate for `max` is:

$$\begin{aligned}&\text{corres}_{L2}\ \text{globals ret\_\_int\_'}\ (\lambda s.\ ())\\&\quad(\lambda s.\ \text{a\_'}\ s = a \wedge \text{b\_'}\ s = b)\ (\text{max}_{L2}\ m\ a\ b)\ (\text{max}_{L1}\ m)\end{aligned}$$

In this example the state translation function globals strips away local variables from the L1 state; the return extraction function *rx* ensures the value returned by max_{L2} matches the variable ret__int_' of max_{L1}, while the exception extraction function *ex* is unused and simply returns unit, as no exceptions are thrown by the `max` function.[3] The remainder of the predicate states that, assuming the inputs a and b to our max_{L2} function match those of the L1 state, then the return value of our max_{L2} function will match the L1 state variable ret__int_' after executing max_{L1}.

[3] The original Simpl function used exceptional control flow to represent the abrupt termination of the `return` statement, but in this case could be optimised away by the exception elimination optimisations described in Section 4.5.2. The only remains of the exceptions are the updates to the phoney `global_exn_var` variable used by the C-to-Isabelle parser to track the reason for the (now eliminated) throw.

$$\frac{}{\mathsf{corres}_{\mathsf{L2}}\ st\ rx\ ex\ P\ \mathsf{skip}_{\mathsf{E}}\ (\mathsf{gets}_{\mathsf{E}}\ (\lambda s.\ ()))}\ \textsc{L2CorresSkip} \qquad \frac{}{\mathsf{corres}_{\mathsf{L2}}\ st\ rx\ ex\ P\ \mathsf{fail}_{\mathsf{E}}\ \mathsf{fail}_{\mathsf{E}}}\ \textsc{L2CorresFail}$$

$$\frac{\forall s.\ P\ s \longrightarrow G'\ s = G\ (st\ s)}{\mathsf{L2corres}\ st\ rx\ ex\ P\ (\mathsf{guard}_{\mathsf{E}}\ G)\ (\mathsf{guard}_{\mathsf{E}}\ G')}\ \textsc{L2CorresGuard}$$

$$\frac{\forall s.\ P\ s \longrightarrow st\ s = st\ (m\ s) \qquad \forall s.\ P\ s \longrightarrow rx\ (m\ s) = f\ (st\ s)}{\mathsf{L2corres}\ st\ rx\ ex\ P\ (\mathsf{gets}_{\mathsf{E}}\ f)\ (\mathsf{modify}_{\mathsf{E}}\ m)}\ \textsc{L2CorresGets}$$

$$\frac{\forall s.\ P\ s \longrightarrow m\ (st\ s) = st\ (m'\ s)}{\mathsf{L2corres}\ st\ rx\ ex\ P\ (\mathsf{modify}_{\mathsf{E}}\ m)\ (\mathsf{modify}_{\mathsf{E}}\ m')}\ \textsc{L2CorresModify}$$

$$\frac{\forall s.\ P\ s \longrightarrow ex\ s = v}{\mathsf{L2corres}\ st\ rx\ ex\ P\ (\mathsf{throw}_{\mathsf{E}}\ v)\ (\mathsf{throw}_{\mathsf{E}}\ ())}\ \textsc{L2CorresThrow}$$

$$\frac{\forall s\ a.\ st\ s = st\ (M\ a\ s)}{\mathsf{corres}_{\mathsf{L2}}\ st\ rx\ ex\ P\ \mathsf{unknown}_{\mathsf{E}}\ (\mathsf{init}_{\mathsf{E}}\ M)}\ \textsc{L2CorresInit}$$

Table 5.2: Basic rules used to prove refinement between the L1 and L2 specifications.

The remainder of this section will look at how we generate and automatically prove equivalence of an initial L2 specification from an L1 specification.

5.1.5 Proving the L2 specification

We prove the $\mathsf{corres}_{\mathsf{L2}}$ predicate by using a syntax-directed set of rules. Table 5.2 shows the rules used to prove refinement of leaf statements between L1 and L2, while Table 5.3 shows the compound rules used.

For each L2 and corresponding L1 statement, the proofs are carried out using the same general method:

1. The appropriate rule is selected for the L1/L2 statements based on their syntax, and the rule is instantiated with the corresponding L1 and L2 statements being proven.
2. The variable *st* is instantiated with the state translation function that strips local variables out of the state variable, leaving just the global state such as the global variables and the heap.
3. The variable *rx* in the chosen rule is then instantiated with a function that extracts variables returned by the L2 specification from the concrete state. For example, if an L2 statement returned the variables (a, b), then *rx* would be instantiated to $(\lambda s.\ (\mathrm{a\_}'\ s,\ \mathrm{b\_}'\ s))$. If the L2 statement doesn't return any variables, *rx* is simply instantiated to $(\lambda s.\ ())$.
4. Similarly, the variable *ex* is instantiated with a function that extracts thrown variables out of the state. Even though most L2 statements do not actually throw an exception, *ex* is still instantiated with the appropriate extraction function as if

$$\frac{\begin{array}{c}\mathsf{corres}_{\mathsf{L}_2}\ st\ rx\ ex\ P\ A\ A' \\ \forall r.\ \mathsf{corres}_{\mathsf{L}_2}\ st\ rx'\ ex\ (P'\ r)\ (B\ r)\ B' \\ \{\!|R|\!\}\ A'\ \{\!|\lambda\_\ s.\ P'\ (rx\ s)\ s|\!\},\ \{\!|\lambda\_\ \_.\ \mathsf{True}|\!\} \\ \forall s.\ R\ s \longrightarrow P\ s\end{array}}{\mathsf{corres}_{\mathsf{L}_2}\ st\ rx'\ ex\ R\ (A \gg\!=_{\mathsf{E}} B)\ (A' \gg\!=_{\mathsf{E}} (\lambda\_.\ B'))}$$
L2CorresBind

$$\frac{\begin{array}{c}\mathsf{corres}_{\mathsf{L}_2}\ st\ rx\ ex\ P\ A\ A' \\ \forall r.\ \mathsf{corres}_{\mathsf{L}_2}\ st\ rx\ ex'\ (P'\ r)\ (B\ r)\ B' \\ \{\!|Q|\!\}\ A'\ \{\!|\lambda\_\ \_.\ \mathsf{True}|\!\},\ \{\!|\lambda\_\ s.\ P'\ (ex\ s)\ s|\!\} \\ \forall s.\ Q\ s \longrightarrow P\ s\end{array}}{\mathsf{corres}_{\mathsf{L}_2}\ st\ rx\ ex'\ Q\ (\mathsf{catch}_{\mathsf{E}}\ A\ B)\ (\mathsf{catch}_{\mathsf{E}}\ A'\ (\lambda\_.\ B'))}$$
L2CorresCatch

$$\frac{\begin{array}{c}\mathsf{corres}_{\mathsf{L}_2}\ st\ rx\ ex\ P\ A\ A' \qquad \mathsf{corres}_{\mathsf{L}_2}\ st\ rx\ ex\ P'\ B\ B' \\ \forall s.\ R\ s \longrightarrow P\ s \qquad\qquad \forall s.\ R\ s \longrightarrow P'\ s \\ \forall s.\ R\ s \longrightarrow c'\ s = c\ (st\ s)\end{array}}{\mathsf{corres}_{\mathsf{L}_2}\ st\ rx\ ex\ R\ (\mathsf{condition}_{\mathsf{E}}\ c\ A\ B)\ (\mathsf{condition}_{\mathsf{E}}\ c'\ A'\ B')}$$
L2CorresCond

$$\frac{\begin{array}{c}\forall r.\ \mathsf{corres}_{\mathsf{L}_2}\ st\ rx\ ex\ (Q\ r)\ (B\ r)\ B' \\ \{\!|\lambda s.\ I\ (rx\ s)\ s|\!\}\ B'\ \{\!|\lambda\_\ s.\ I\ (rx\ s)\ s|\!\},\ \{\!|\lambda\_\ \_.\ \mathsf{True}|\!\} \\ \forall s.\ I\ (rx\ s)\ s \longrightarrow c'\ s = c\ (rx\ s)\ (st\ s) \\ \forall s\ r.\ I\ r\ s \longrightarrow Q\ r\ s \qquad \forall s.\ I\ r\ s \longrightarrow rx\ s = r \qquad \forall s.\ P\ r\ s \longrightarrow I\ r\ s\end{array}}{\mathsf{corres}_{\mathsf{L}_2}\ st\ rx\ ex\ (P\ r)\ (\mathsf{whileLoop}_{\mathsf{E}}\ c\ B\ r)\ (\mathsf{whileLoop}_{\mathsf{E}}\ (\lambda\_.\ c')\ (\lambda\_.\ B')\ ())}$$
L2CorresWhile

$$\frac{\begin{array}{c}\forall m.\ \mathsf{corres}_{\mathsf{L}_2}\ st\ rx'\ ex'\ P'\ (f\ m)\ (f'\ m) \\ \forall s\ r.\ st\ (return\_xf\ s\ (teardown\ r\ s)) = st\ s \\ \forall s\ r.\ rx\ (return\_xf\ s\ (teardown\ r\ s)) = rx'\ s \\ \forall s.\ st\ (setup\ s) = st\ s \qquad \forall s.\ P\ s \longrightarrow P'\ (setup\ s)\end{array}}{\begin{array}{l}\mathsf{corres}_{\mathsf{L}_2}\ st\ rx\ ex\ P\ (\mathsf{call}_{\mathsf{L}_2}\ (f\ m)) \\ \quad (\mathsf{call}_{\mathsf{L}_1}\ setup\ (f'\ m)\ teardown\ return\_xf)\end{array}}$$
L2CorresCall

Table 5.3: Compound rules used to prove refinement between the L1 and L2 specifications.

the statement *could* throw an exception. This is because every statement in the left-hand side of a catch_E block must have an identical exception type in order to be type correct.

5. The precondition P of the rule is instantiated to ensure that, for each local variable read by the current statement, the L2 bound variables match the L1 variables in the state. For example, if the L2 statement being proven was

$$\text{gets}_E\ (\lambda s.\ a\ +\ b\ +\ c)$$

which reads the set of variables $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, then the variable P would be instantiated with the predicate

$$\lambda s.\ a = \mathtt{a\_}'\ s \wedge b = b\_'s \wedge c = \mathtt{c\_}'\ s$$

6. In the case of `while` loops, an invariant I is generated stating that local variables that are live between loop iterations in the L2 specification match their L1 equivalents.
7. If the rule is a compound rule (i.e., it contains child blocks of code), a proof is recursively produced for the child blocks. Any corres_{L_2} assumptions in the rule are instantiated with these proofs.
8. Finally, any remaining side-conditions are automatically discharged, as explained below.

The majority of these steps are to set up the proofs that ensure that the bound variables of the L2 specification correspond to the local variables in the L1 state.

In the final step of the above process, AutoCorres discharges any side-conditions arising from the use of the corres_{L_2} ruleset. The rules L2CorresSkip and L2CorresFail do not have any side-conditions to be discharged, so apply without further effort. Rules that read from the state, such as L2CorresGuard and L2CorresModify have a side condition that ensures that the value of the expression does not change when abstracting from L1 to L2. Rules that return a value, such as L2CorresGets and L2CorresThrow, have a side-condition that ensures that the variables returned by the L2 statement match those in the L1 state, using extraction functions *rx* or *ex*. These side-conditions can be discharged using Isabelle/HOL's simplifier.

The compound rules L2CorresBind, L2CorresCatch and L2CorresWhile have an additional side-condition showing that a Hoare triple holds. This side-condition is a *preservation condition*, where we must show that the child blocks preserve the values of variables that are required by later blocks. For instance, in the code fragment

$$\begin{array}{l}\textbf{do}\ \text{modify}_E\ (\lambda s.\ s(\!|\, \mathtt{a\_}' := 1 \,|\!));\\ \quad \text{modify}_E\ (\lambda s.\ s(\!|\, \mathtt{b\_}' := \mathtt{c\_}'\ s \,|\!))\\ \textbf{od}\end{array}$$

the right-hand side of the bind depends on the value of `c`. When the left-hand side is translated into L2 format it will return a new value of `a`, but will not return a new value of `c` because it (purportedly) didn't change its value. The preservation condition

is required to prove that this actually is the case. For instance, in this example the preservation side-condition would be

$$\{\!|\lambda s.\ \mathrm{c\_}'\ s = c|\!\}\ \mathsf{modify}_{\mathsf{E}}\ (\lambda s.\ s(\!|\mathrm{a\_}' := 1|\!))\ \{\!|\lambda rv\ s.\ \mathrm{c\_}'\ s = c|\!\},\ \{\!|\lambda rv\ s.\ \mathsf{True}|\!\}$$

Such preservation conditions are automatically discharged using a simple syntax-directed ruleset.

Finally, the rule L2CorresCall has some additional side conditions that show that (*i*) setting up the callee *f*'s scope doesn't affect the L2 state; (*ii*) tearing down or extracting the return value out of the callee *f*'s scope doesn't affect the L2 state; (*iii*) the extraction function *rx* still returns the same set of variables as the callee *f* after shuffling the state through *return_xf* and *teardown*; and finally (*iv*) if the L1 state satisfies the precondition *P*, then running *setup* will cause the state to now satisfy the callee's precondition *P*′. At a high level, all of these side-conditions ensure that the parameters *setup*, *teardown* and *return_xf* generated by the C-to-Isabelle parser are well-behaved—only changing local variables in a way reflected in the L2 specification's use of bound variables—and hence can be thrown away in the generated L2 specifications. Each of these side-conditions can be discharged by Isabelle/HOL's simplifier.

At the end of the process described above, AutoCorres will have produced a $\mathsf{corres}_{\mathsf{L2}}$ theorem stating that the original L1 input specification is a refinement of the generated L2 output specification, assuming that arguments to the generated L2 function match the corresponding local variables in the L1 state.

Both the generation of the L2 specification and its proof are completely automatic, and hidden from the end-user. In the hypothetical case that the generated L2 translation is incorrect, the proof process will fail with an internal error. In this case, the end-user is responsible for approaching the desk of the AutoCorres authors and bitterly complaining. The end-user will have some comfort, however, in knowing that if AutoCorres completes the translation, then it is correct.

5.2 Further program optimisations

A significant benefit of lifted local form is that it allows us to easily determine how local variables are used, and carry out simplifications based on this. Such simplifications include

- Removing code that writes to a local variable never subsequently read from:

$$(\mathsf{gets}_{\mathsf{E}}\ g\ \gg\!=_{\mathsf{E}}\ (\lambda\_.\ f)) = f$$

- Similarly, removing code that initialises local variables to non-deterministically chosen values using $\mathsf{unknown}_{\mathsf{E}}$ when we can prove that the value won't be read from:

$$(\mathsf{unknown}_{\mathsf{E}}\ \gg\!=_{\mathsf{E}}\ (\lambda\_.\ f)) = f$$

- Using assumptions from $\mathsf{guard}_{\mathsf{E}}$, $\mathsf{condition}_{\mathsf{E}}$ and $\mathsf{whileLoop}_{\mathsf{E}}$ statements to simplify later expressions; and

```
max_L2 m' a b ≡
    do ret ← unknown_E;
       ret ← condition_E (λs. a ≤_s b)
                 (do ret ← gets_E (λs. b);
                     global_exn_var ← gets_E (λ_. Return);
                     gets_E (λs. ret)
                  od)
                 (do ret ← gets_E (λs. a);
                     global_exn_var ← gets_E (λ_. Return);
                     gets_E (λs. ret)
                  od);
       return_E ret
    od
```

(a) Function `max` translated to L2

```
max_L2 m' a b ≡
    condition_E (λs. a ≤_s b)
        (gets_E (λs. b))
        (gets_E (λs. a))
```

(b) Function `max` after flow-sensitive optimisations

Figure 5.10: The `max` function shown after translation to L2, and after flow-sensitive optimisations have taken place.

- Collapsing variables that are only used once into the locations where they are used.

By allowing constant-valued expressions to be folded into the location they are used, we are also able to discharge many more guard_E statements not previously provable. AutoCorres additionally repeats the peephole optimisations carried out at the end of the Simpl-to-L1 conversion, described in Section 4.5.1, as many of the flow-sensitive optimisations enable new uses of the L1 peephole optimisations, and *vice versa*.

Figure 5.10 shows the `max` function after flow-sensitive optimisations. The redundant unknown_E statement and variable global_exn_var are discarded. The two gets_E terms in each branch of the condition_E are also collapsed into a single statement. Finally, the last, unnecessary return_E statement is discarded. The result is a much simpler program.

5.3 Type strengthening

So far, all of our generated programs have been written using Cock et al.'s exception monad. Section 4.2.1 outlined some of the motivations for using this monad, including our aim to model C code that reads and writes to global state, has abrupt termination, uses non-determinism, or could possibly fail.

For the majority of the code we translate, many of these features are not required. For example, Section 4.5.2 describes how the majority of exception usage can be

eliminated from specifications. The use of non-determinism is mostly limited to setting up uninitialised variables, many of which are eliminated using the simplifications in Section 5.2 above. Further, many functions do not modify the state of the system at all, either only reading the global state or having results that depend entirely on their input parameters. In these cases, the exception monad is far more expressive than required. Less expressive monads constrain program by type, giving the user 'free' theorems. For instance, users can avoid having to reason that a particular function preserves a program-wide invariant if that function doesn't accept the program state as a parameter.

We therefore specialise the type of individual functions to contain only the features they require. The types we use are as follows, in decreasing strength:

Pure functional These are standard Isabelle functions, where the function returns a deterministic output depending only on its input parameters (and read-only access to the state which, if necessary, will be passed in as an additional parameter).

In these cases, monadic bind statements can be replaced with a simple functional let construct and monadic $\mathsf{condition}_\mathsf{E}$ statements with a simple if statement. Our example `max` function falls into this category.

Option monad The C standard is littered with restrictions that result in $\mathsf{guard}_\mathsf{E}$ statements that cannot be automatically discharged. Unfortunately, even a single such $\mathsf{guard}_\mathsf{E}$ statement will prevent a function from being translated into a pure Isabelle function, as we must consider failed executions.

We can, however, use the *option monad*, where every computation either results in a single value a (represented as Some a), or failure (represented as None). Any intermediate failure results in failure of the entire computation. The option monad defines monadic functions $\mathsf{return}_\mathsf{o}$, fail_o, gets_o, and so on, analogous to their exception monad counterparts $\mathsf{return}_\mathsf{E}$, fail_E, and gets_E. The full definitions are shown in Table 5.4.

Functions that may potentially fail but are deterministic, have simple control flow, and only read from global state can be transformed into the option monad. Callers of such functions will translate a result of None into failure.

State monad Functions that need to modify global state or use non-determinism but do not use exceptional control flow are translated into a state monad, without support for exceptions.

Type strengthening takes place using a series of rewrite rules that attempt to strengthen individual parts of the program starting at the leaves, and then combine partial results to strengthen larger parts of the program. The rules used are shown in Table 5.5 and Table 5.6.

Program fragments using the strengthened types can be embedded in our exception monad using a *lifting function* for the type, also shown in Table 5.5 and Table 5.6. For example, the option monad is embedded in the exception monad using the lifting function $\mathsf{gets\_the}_\mathsf{E}$, defined as follows:

$$
\begin{aligned}
&\mathsf{gets\_the}_\mathsf{E}\ f \equiv \\
&\quad \textbf{do}\ v \leftarrow \mathsf{gets}_\mathsf{E}\ f; \\
&\qquad \mathsf{guard}_\mathsf{E}\ (\lambda\_.\ v \neq \mathsf{None}); \\
&\qquad \mathsf{return}_\mathsf{E}\ (\mathsf{the}\ v) \\
&\quad \textbf{od}
\end{aligned}
$$

$$(f \gg=_o g) \equiv \lambda s.\ \text{case}\ f\ s\ \text{of None} \Rightarrow \text{None} \mid \text{Some}\ x \Rightarrow g\ x\ s$$

$$\text{return}_o\ x \equiv \lambda s.\ \text{Some}\ x$$

$$\text{fail}_o \equiv \lambda s.\ \text{None}$$

$$\text{gets}_o\ f \equiv \lambda s.\ \text{Some}\ (f\ s)$$

$$\text{condition}_o\ c\ L\ R \equiv \lambda s.\ \text{if}\ c\ s\ \text{then}\ L\ s\ \text{else}\ R\ s$$

$$\text{guard}_o\ G \equiv \lambda s.\ \text{if}\ G\ s\ \text{then Some}\ ()\ \text{else None}$$

Table 5.4: Definitions of the basic option monad primitives.

Pure functional

$$\underset{\textit{Lifting Function}}{\text{return}_E} \qquad \underset{\textit{Call Rule}}{\frac{f = \text{return}_E\ f'}{\text{call}_{L2}\ f = \text{return}_E\ f'}}$$

$$(\textbf{do}\ a \leftarrow \text{return}_E\ A;\ \text{return}_E\ (B\ a)\ \textbf{od}) = \text{return}_E\ (\text{let}\ a = A\ \text{in}\ B\ a) \quad \text{TsPureBind}$$

$$\text{condition}_E\ (\lambda\_.\ c)\ (\text{return}_E\ A)\ (\text{return}_E\ B) = \text{return}_E\ (\text{if}\ c\ \text{then}\ A\ \text{else}\ B) \quad \text{TsPureCond}$$

Option monad

$$\underset{\textit{Lifting Function}}{\text{gets\_the}_E} \qquad \underset{\textit{Call Rule}}{\frac{f = \text{gets\_the}_E\ f'}{\text{call}_{L2}\ f = \text{gets\_the}_E\ f'}}$$

$$\text{gets}_E\ a = \text{gets\_the}_E\ (\text{gets}_o\ a) \quad \text{TsOptGets}$$

$$\text{guard}_E\ G = \text{gets\_the}_E\ (\text{guard}_o\ G) \quad \text{TsOptGuard}$$

$$\text{fail}_E = \text{gets\_the}_E\ \text{fail}_o \quad \text{TsOptFail}$$

$$\text{gets\_the}_E\ X \gg=_E (\lambda s.\ \text{gets\_the}_E\ (Y\ s)) = \text{gets\_the}_E\ (X \gg=_o Y) \quad \text{TsOptBind}$$

$$\text{condition}_E\ C\ (\text{gets\_the}_E\ L)\ (\text{gets\_the}_E\ R) = \text{gets\_the}_E\ (\text{condition}_o\ C\ L\ R) \quad \text{TsOptCond}$$

$$\text{whileLoop}_E\ C\ (\lambda x.\ \text{gets\_the}_E\ (B\ x))\ i = \text{gets\_the}_E\ (\text{while}_o\ C\ B\ i) \quad \text{TsOptWhile}$$

Table 5.5: Type strengthening rules used to convert specifications from the exception monad to pure functional types and the option monad.

State monad

$$\underset{\textit{Lifting Function}}{\mathsf{lift}_E} \qquad \underset{\textit{Call Rule}}{\frac{f = \mathsf{lift}_E\ f'}{\mathsf{call}_{L_2}\ f = \mathsf{lift}_E\ f'}}$$

$$\underset{\textsc{TsStGets}}{\mathsf{gets}_E\ a = \mathsf{lift}_E\ (\mathsf{gets}\ a)} \qquad \underset{\textsc{TsStModify}}{\mathsf{modify}_E\ m = \mathsf{lift}_E\ (\mathsf{modify}\ m)}$$

$$\underset{\textsc{TsStUnknown}}{\mathsf{unknown}_E = \mathsf{lift}_E\ \mathsf{unknown}} \qquad \underset{\textsc{TsStGuard}}{\mathsf{guard}_E\ G = \mathsf{lift}_E\ (\mathsf{guard}\ G)} \qquad \underset{\textsc{TsStFail}}{\mathsf{fail}_E = \mathsf{lift}_E\ \mathsf{fail}}$$

$$\underset{\textsc{TsStSeq}}{\mathsf{lift}_E\ A \gg\!=_E (\lambda r.\ \mathsf{lift}_E\ (B\ r)) = \mathsf{lift}_E\ (A \gg\!= B)}$$

$$\underset{\textsc{TsStCond}}{\mathsf{condition}_E\ c\ (\mathsf{lift}_E\ A)\ (\mathsf{lift}_E\ B) = \mathsf{lift}_E\ (\mathsf{condition}\ c\ A\ B)} \qquad \underset{\textsc{TsStWhile}}{\mathsf{whileLoop}_E\ c\ (\lambda r.\ \mathsf{lift}_E\ (B\ r))\ i = \mathsf{lift}_E\ (\mathsf{whileLoop}\ c\ B\ i)}$$

$$\underset{\textsc{TsStCatch}}{\mathsf{catch}_E\ (\mathsf{lift}_E\ A)\ B = \mathsf{lift}_E\ A} \qquad \underset{\textsc{TsStCatchConv}}{\mathsf{catch}_E\ A\ (\lambda r.\ \mathsf{lift}_E\ (B\ r)) = \mathsf{lift}_E\ (\mathsf{catch}\ A\ B)}$$

Table 5.6: Type strengthening rules used to convert specifications from the exception monad to a plain non-deterministic state monad where possible.

$$\begin{aligned}&\mathsf{max}_{L_2}\ m'\ a\ b \equiv \\ &\quad \mathsf{condition}_E\ (\lambda s.\ a \leq_s b) \\ &\quad\quad (\mathsf{return}_E\ b) \\ &\quad\quad (\mathsf{return}_E\ a)\end{aligned}$$

(a) Exception Monad

$$\begin{aligned}&\mathsf{max}_{L_2}\ m'\ a\ b \equiv \\ &\quad \mathsf{lift}_E\ (\mathsf{condition}\ (\lambda s.\ a \leq_s b) \\ &\quad\quad (\mathsf{return}\ b) \\ &\quad\quad (\mathsf{return}\ a))\end{aligned}$$

(b) State Monad

$$\begin{aligned}&\mathsf{max}_{L_2}\ m'\ a\ b \equiv \\ &\quad \mathsf{gets\_the}_E\ (\mathsf{condition}_o\ (\lambda s.\ a \leq_s b)\ (\mathsf{return}_o\ b)\ (\mathsf{return}_o\ a))\end{aligned}$$

(c) Option Monad

$$\begin{aligned}&\mathsf{max}_{L_2}\ m'\ a\ b \equiv \\ &\quad \mathsf{return}_E\ (\mathsf{if}\ a \leq_s b\ \mathsf{then}\ b\ \mathsf{else}\ a)\end{aligned}$$

(d) Pure Functional

Figure 5.11: The `max` function represented in four different types.

This function converts a None result of the option monad into a failure result in the exception monad, and converts a Some x result into a monadic return value. A call to a function f in the option monad will be converted to $\text{gets\_the}_\text{E}\ f$.

For each target type, the actual rewriting process proceeds as follows:

1. We start by applying the set of type-strengthening rewrite rules for the target type to the body of the function being strengthened. Figure 5.11 shows the max_{L_2} function in its original form, and rewritten with each set of rules.
2. Once no more rules in the current set can be applied, we check if the rewritten function takes the form $f = L\ f'$, where f is the old definition of the function, L is the lifting function for the type we are attempting to strengthen to, and f' is the body of the function in its type-strengthened form. If the rewritten function has this form, then type-strengthening has succeeded; if not, then the function cannot be automatically strengthened to the current type, and we try another.
3. Given the function is now in the form $f = L\ f'$, we define a new constant for the function with a definition of just f'. We additionally instantiate the *calling theorem* of the target type to generate a theorem of the form

$$\text{call}_{\text{L}_2}\ f\ =\ L\ f'$$

which allows functions calls to f to be be rewritten as $L\ f'$.

For example, Figure 5.11(a) shows the max_{L_2} function in its original exception monad type, while Figure 5.11(b–d) show the function rewritten using the three different sets of type strengthening rules. All three rewritten equations for max_{L_2} have the form $f = L\ f'$ where L is the lifting function lift_E, gets_the_E or return_E. We can now define a new function max_{TS} containing just the body of the pure functional version of max_{L_2}, giving us the function definition

$$\begin{aligned}&\text{max}_{\text{TS}}\ m'\ a\ b \equiv \\ &\qquad \text{if}\ a \leq_s b\ \text{then}\ b\ \text{else}\ a\end{aligned}$$

Additionally, we can derive a calling theorem for this rule allowing calls to max_{L_2} to be rewritten to max_{TS}:

$$\text{call}_{\text{L}_2}\ (\text{max}_{\text{L}_2}\ m'\ a\ b)\ =\ \text{return}_\text{E}\ (\text{max}_{\text{TS}}\ m'\ a\ b)$$

To determine which type we can strengthen each function to, we attempt to apply each set of strengthening rules in order from strongest type to weakest type. If a particular function can be converted to the type, then we are finished. Otherwise, we continue to try alternative, more expressive representations. The final translation to the state monad is 'best-effort': it will convert as much of each function as possible, but will resort to using the rule TsStCatchConv, embedding fragments of the function that couldn't be strengthened out of the exception monad in the function using the catch constant.

Functions are converted in topological order based on their call-graph. If a function f calls a second function g, f will need a type at least as expressive as g. Mutually

Table 5.7: Number of functions in the seL4 microkernel translated into each type. Percentages do not add up to 100% due to rounding.

Type	Count	Percent
Pure function	151	28.2%
Option monad	51	9.6%
State monad	309	57.8%
Exception monad	24	4.5%
Total	535	

recursive functions are converted by attempting to convert each function in the group individually, and then selecting the weakest type required by any of the functions.

AutoCorres performs type strengthening as its very last abstraction step. With the exception of some final *polishing* simplifications described below, carrying out type strengthening last prevents intermediate phases of AutoCorres from needing to support a variety of different input types, focusing instead on just the exception monad. In particular, the word abstraction and heap abstraction phases of AutoCorres that will be described in Chapter 6 and Chapter 7 occur *before* type strengthening. An image depicting the order that translations are carried out is shown in Figure 1.2, on page 6.

Type strengthening is effective in practice. Table 5.7 shows statistics of type strengthening used on the seL4 microkernel source code, with almost 96% of functions being strengthened into another type. The remaining 4% are functions that could not be rewritten to avoid using exceptions in the exception elimination optimisations described in Section 4.5.2.

5.4 Polishing and final theorem

Our final translation we term *polishing*, where transformations that make the specification easier to read for humans at the cost of being harder to perform further automatic transformations on are carried out. Examples of such transformations include adding syntax annotations to improve the on-screen display of output specifications, simplifying condition statements (for instance, translating condition $c\ A$ skip to the equivalent, but shorter, statement when $c\ A$), and other minor transformations that improve readability of the output.

In this final stage, we also remove the recursive measure parameters added to every function in Section 4.4.2 when we can prove they are not necessary (i.e., if there is no reference to them in the function's body).

The final output of the `max` function after the polish phase is simply

$$\mathsf{max}'\ a\ b \equiv \mathsf{if}\ a \leq_s b\ \mathsf{then}\ b\ \mathsf{else}\ a$$

Other than the type of the arguments, this precisely matches the definition of Isabelle/HOL's built-in max definition:[4]

$$\mathsf{max}\ a\ b = (\mathsf{if}\ a \leq b\ \mathsf{then}\ b\ \mathsf{else}\ a)$$

In this final phase, AutoCorres also collects the theorems that show refinement between the various intermediate phases of AutoCorres, and combines them to form a final theorem linking the input Simpl to the output.

The final theorem is a predicate ccorres, defined as follows:

$$\begin{array}{l}\mathsf{ccorres}\ st\ \Gamma\ rx\ P\ A\ C \equiv \\ \quad \forall s.\ P\ s \wedge \neg\ \mathsf{failed}\ (A\ (st\ s)) \longrightarrow \\ \quad\quad (\forall t.\ \Gamma \vdash \langle C,\ \mathsf{Normal}\ s\rangle \Rightarrow t \longrightarrow \\ \quad\quad\quad (\exists s'.\ t = \mathsf{Normal}\ s' \wedge \\ \quad\quad\quad\quad (\mathsf{Norm}\ (rx\ s'),\ st\ s') \in \mathsf{results}\ (A\ (st\ s)))) \wedge \\ \quad\quad \Gamma \vdash C \downarrow \mathsf{Normal}\ s\end{array}$$

This predicate is the combination of the $\mathsf{corres}_{\mathsf{L1}}$ and $\mathsf{corres}_{\mathsf{L2}}$ predicates, and states that for all Simpl states matching the precondition P, assuming that the program A executing from state $st\ s$ does not fail, then the following holds:

1. For each normal execution of C there is an equivalent execution of A whose return value will match the value extracted using rx from C's state;
2. The execution of C will not fail, terminate abruptly (i.e., finish in an exceptional state), or get stuck; and
3. The execution of C will terminate.

The ccorres predicate is proven automatically by piecing together intermediate theorems generated in each phase and using the rule CCorresChain, defined as follows:

$$\frac{\begin{array}{c}\mathsf{corres}_{\mathsf{L1}}\ \Gamma\ A_{L1}\ A_C \\ \mathsf{corres}_{\mathsf{L2}}\ st_{L2}\ rx_{L2}\ ex_{L2}\ P_{L2}\ A_{L2}\ A_{L1} \\ A_{TS} = \mathsf{call}_{\mathsf{L2}}\ A_{L2}\end{array}}{\mathsf{ccorres}\ st_{L2}\ \Gamma\ rx_{L2}\ P_{L2}\ A_{TS}\ A_C}$$

The $\mathsf{corres}_{\mathsf{L1}}$ and $\mathsf{corres}_{\mathsf{L2}}$ assumptions are discharged from the theorems originating from the Simpl-to-L1 and L1-to-L2 phases of AutoCorres, respectively; while the final assumption $A_{TS} = \mathsf{call}_{\mathsf{L2}}\ A_{L2}$ is the theorem generated from the type strengthening phase.

We don't consider the generated ccorres predicate a useful theorem in and of itself, but rather it merely presents a starting point for the *real* verification work the user

[4] We return to this example again in Chapter 6, where we use *word abstraction* to further transform the `max` function so that it precisely matches Isabelle/HOL's max function, including the type.

Table 5.8: Lines of specification and average term size after each translation phase of the seL4 source [81]. Both the absolute number and the number relative to the original Simpl input are provided. The *Heap Abstraction* translation refers to the phase described in Chapter 7, while *Word Abstraction* refers to the phase described in Chapter 6.

Specification	Lines of Spec		Avg. Term Size	
	Absolute	Relative	Absolute	Relative
Simpl	20 654	100.0%	304	100.0%
Shallow Embedding	36 955	178.9%	327	107.6%
Control-Flow Peephole	26 075	126.2%	255	83.9%
Exception Elimination	24 712	119.6%	248	81.6%
Lifted Local Vars	25 017	121.1%	202	66.4%
Flow-Sensitive Opts.	13 570	65.7%	145	47.7%
Heap Abstraction	12 846	62.2%	118	38.8%
Word Abstraction	12 743	61.7%	118	38.8%
Type Strengthening	12 674	61.4%	115	37.8%
Polish	11 609	56.2%	109	35.9%

needs to carry out. For example, a Hoare-style proof on a Simpl function can be lifted to a proof on the output of AutoCorres, using the rule

$$\frac{\begin{array}{c}\mathsf{ccorres}\ st\ G\ rx\ P'\ A\ C \\ \{\!|\ P\ |\!\}\ A\ \{\!|\ Q\ |\!\},\ \{\!|\ \lambda rv\ s.\ \mathsf{True}\ |\!\} \\ \forall s.\ P\ (st\ s) \longrightarrow P'\ s\end{array}}{G,\ \Theta \vdash_{t/F} \{s.\ P\ (st\ s)\}\ C\ \{s.\ Q\ (rx\ s)\ (st\ s)\},\ E}$$

Here, a total-correctness proof using Schirmer's Simpl framework [91, 93] can be lifted into a monadic Hoare-based proof using the generated ccorres predicate from AutoCorres.

The generated ccorres predicate can also be used as a stepping stone towards many other goals, such as refinement to an even higher-level specification [81, 86], showing equivalence between two programs, or as a starting point for further automated analysis. We explore such uses in Chapter 8, but for the moment just point out that such reasoning can be carried out.

5.5 Conclusion

In this chapter, we have presented a method of automatically translating the suboptimal L1 monadic program representation we generated in the last chapter into much simpler forms. Lifting local variables out of the program's global state, modelling them instead using monadic bound variables, both simplifies reasoning and enables our further flow-sensitive optimisations and type strengthening steps.

After carrying out these steps, our program representations become significantly simpler than their original Simpl equivalents. Table 5.8 shows the different phases of AutoCorres running on the seL4 microkernel's source code [81], and the effect the phases have on the size of the program's logical representation. Two metrics are given. The first is *lines of specification*, which is the number of lines generated when displaying the term using Isabelle/HOL's pretty printer. The second is *average term size*, which is the average number of nodes in each function's abstract syntax tree when parsed by Isabelle/HOL. While neither metric is a perfect measure for specification complexity, the values in the table match our intuition that the output of AutoCorres is simpler than its Simpl input.

There are still two major frustrations that arise when attempting to reason about larger programs, however. Reasoning about programs that carry out word arithmetic still remains problematic, as does reasoning about programs that interact with the system heap. In the next two chapters, we look at how we can address these issues, further easing program verification.

Chapter Summary

- The L1 monadic representation of programs that we generated in the previous chapter models local variables as being stored in the program's global state. Such a representation is clumsy to reason about.
- We lift local variables out of the program's state, modelling them as monadic bound variables. We call this new monadic representation *L2*. The transformation is automatic and generates a proof in Isabelle/HOL of its correctness.
- The new L2 representation enables further program simplifications, where values of local variables that are known earlier in a function can be used to simplify expressions later in the function. This optimisation significantly reduces the size of our program representations.
- Finally, we carry out *type strengthening* on programs, where Cock et al.'s complex exception monad is replaced with a simpler type where possible, such as a standard state monad, an option monad, or simply just a pure Isabelle/HOL function.

6 Word abstraction

In the previous chapters, we demonstrated how programs can be transformed from deeply embedded Simpl representations to more convenient monadic representations. While these transformations ease reasoning, certain aspects of verifying real C programs remain difficult. One of those problems is the difficulty of reasoning about C programs that carry out word-based arithmetic. In this chapter we look at how we can automatically and verifiably abstract such programs into representations that operate on unbounded integers and naturals.

We start in Section 6.1 by describing the difficulties associated with reasoning about word types in C. An informal description of our proposed approach of abstracting finite word-based types into unbounded numbers (and an argument as to why this is sound) can be found in Section 6.2.

In Section 6.3 we develop a formal definition of what it means for such an abstraction to be correct, and also present a user-extensible set of rules that allows us to automatically abstract L2 programs while simultaneously generating a proof of correctness. We conclude by providing a number of smaller case studies in Section 6.4 and Section 6.5.

This chapter is based on the published work by Greenaway et al. [50], *Don't sweat the small stuff: formal verification of C code without the pain* in PLDI 2014.

6.1 Reasoning about word arithmetic

As a low-level language, C makes little attempt to hide details of hardware arithmetic from the programmer. For instance, on a 32-bit system, the range of the signed `int` datatype is -2^{31} to $2^{31} - 1$, while the range of `unsigned int` is 0 to $2^{32} - 1$. An *overflow* occurs when the result of a calculation falls out of this range.

The C standard [55] dictates different behaviours for signed and unsigned datatypes when overflow occurs. For unsigned datatypes, the result of the operation is simply

Incorrect Equation	Counter-example	
$s = s + 1 - 1$	$s = 2^{31} - 1$	*(undefined behaviour)*
$s = -(-s)$	$s = -2^{31}$	*(undefined behaviour)*
$u + 1 > u$	$u = 2^{32} - 1$	*(incorrect result)*
$2 \times u = 4 \longrightarrow u = 2$	$u = 2^{31} + 2$	*(incorrect result)*
$-u = u \longrightarrow u = 0$	$u = 2^{31}$	*(incorrect result)*

Table 6.1: Examples of incorrect mathematical reasoning in C. Variable s is a 32-bit signed `int`, while u is a 32-bit `unsigned int`.

calculated modulo $M + 1$, where M is the maximum value that can be represented in the type. For example, for a 32-bit type, $2^{31} \times 2 = 0$. Signed arithmetic has stricter rules. The C standard states that it is *undefined behaviour* for a program to perform signed arithmetic that overflows: the compiler is free to assume that such behaviour will never occur and, if it does occur, is free to exhibit any behaviour it desires.[1] In modern C compilers, this is not merely an academic issue: for instance, `gcc-4.7` will happily optimise the signed expression `s + 1 > s` to `true` [105].

In the context of program verification, this means that a program specification must precisely model unsigned overflow, and ensure that signed arithmetic operations will not overflow. Norrish's C parser ensures this by translating variables to Isabelle/HOL's finite word types; `unsigned int`'s are translated into the unsigned *word32* type, while `int`'s are translated into the signed *sword32* type. Additionally, for signed operations, the C parser emits guard statements to check that the result does not overflow. For example, the signed C expression `a + b` is translated into:

```
do guard (λs. INT_MIN ≤ sint a + sint b);
   guard (λs. sint a + sint b ≤ INT_MAX);
   return (a +s b)
od
```

Here, the function sint of type *sword32* ⇒ *int* converts the finite 32-bit signed word type into the unbounded Isabelle/HOL integer type. The analogous function unat similarly converts unsigned words into natural numbers, and is used for unsigned expressions. We use the suffix "*s*" to indicate that the operation is being carried out on signed words (for example, $+_s$), and use the suffix "*w*" for unsigned word arithmetic.

While this approach allows C arithmetic to be correctly modelled, actually *reasoning* about it remains burdensome. Table 6.1 lists some 'obvious' mathematical identities that are simply not true when reasoning about C programs. Further, while Isabelle/HOL contains extensive libraries of theorems about natural numbers and integers, these theorems cannot be used when verifying C programs. Large verification projects also

[1] The rationale for preventing programs from triggering signed overflow is to allow C programs to correctly run on old or specialised hardware, such as one's-complement CPUs, CPUs that trap on signed overflow, or CPUs that use saturating arithmetic (such as many audio DSPs).

experience the burden of word-proofs: approximately 25% of the 30 000 lines of proof library developed in the seL4 project [57] were dedicated to word arithmetic theorems.[2]

We are not the only ones to observe the burden of word proofs. Noschinski et al. [86] note that in the verification of graph algorithms, a large amount of their non-algorithmic reasoning went to discharging simple word proofs. They go on to say "[t]his was somewhat surprising, because the only arithmetic operations occurring in the program are equality and increment against a fixed upper bound".

It is clear that we need a better approach to reasoning about word arithmetic.

6.2 Word abstraction

Ideally, we would like to abstract *word32* and *sword32* data types into unbounded natural numbers and integers, respectively. This would avoid the corner cases described above, and also allow Isabelle/HOL's existing proof libraries to be freely used in program proofs. The question is: how can this be done in a sound manner? We can't simply pretend that the underlying hardware can perform arithmetic on arbitrarily large numbers, nor can we ignore C's requirement that signed arithmetic never overflows—or can we?

We observe that verification engineers must *already* prove that signed arithmetic doesn't fall out of the range -2^{31} to $2^{31}-1$, because the C standard demands it. The C parser already inserts corresponding proof obligations. We can thus abstract *sword32* types into *int* types, utilising the existing guard statements to know that the abstract values will always remain in the range of representable values at the concrete level.

Unsigned arithmetic is slightly more difficult; the Simpl program will not contain any guards to ensure that overflow doesn't occur, and more importantly, the source C program may actually *rely* on overflow to occur. Despite this, for many functions unsigned overflow is not expected and—if the program verifier is willing to prove that it does not occur by having additional guard statements in their abstracted output—we can abstract unsigned arithmetic to natural numbers. We allow the user to select whether to use word abstraction or not on a per-function basis.

An example where such abstraction makes sense is in a binary search that calculates the middle element of an array:

```
unsigned int m = (l + r) / 2;
```

A typical verification condition that arises is showing that the selected element remains between the elements l and r:

$$l <_w r \longrightarrow l \leq_w (l +_w r) \text{ div}_w 2 \wedge (l +_w r) \text{ div}_w 2 <_w r$$

If the terms l, m and r were of type *nat*, this theorem is solved automatically using Isabelle/HOL's built-in `auto` tactic. On the original *word32* type, however, an additional

[2] Such theorems included reasoning about when word arithmetic does and doesn't overflow, how operations on words correspond to their unbounded and integer equivalents, how multiple smaller word values can be packed and unpacked into a single larger word value, on so on.

precondition unat l + unat $r < 2^{32}$ is required and the proof term must be manually lifted into the naturals before it can finally be solved using existing theorems in Isabelle/HOL's library.[3] If we could convert the latter rule to the former rule, verification of programs containing word-based arithmetic would be greatly simplified.

6.3 Performing the abstraction

Our implementation of word abstraction converts local variables and arguments of functions, but does not attempt to modify values stored in memory or global variables. Instead, expressions of type *nat* and *int* are cast back to their machine-word equivalents when written to memory, and *vice versa*. This means that the program's state remains unmodified, and the abstraction process only has to adjust expressions in the program.

We generate a refinement theorem showing that the input program C refines our word-abstracted program A:

$$
\begin{array}{l}
\mathsf{corres}_{\mathsf{WA}}\ P\ rx\ ex\ A\ C \equiv \\
\quad \forall s.\ P\ s \wedge \neg\ \mathsf{failed}\ (A\ s) \longrightarrow \\
\quad\quad (\forall (r,\ t) \in \mathsf{results}\ (C\ s). \\
\quad\quad\quad \mathsf{case}\ r\ \mathsf{of} \\
\quad\quad\quad\quad \mathsf{Exc}\ v \Rightarrow (\mathsf{Exc}\ (ex\ v),\ t) \in \mathsf{results}\ (A\ s) \\
\quad\quad\quad\quad \mid \mathsf{Norm}\ e \Rightarrow (\mathsf{Norm}\ (rx\ e),\ t) \in \mathsf{results}\ (A\ s)) \wedge \\
\quad\quad \neg\ \mathsf{failed}\ (C\ s)
\end{array}
$$

The precondition P states under which conditions our $\mathsf{corres}_{\mathsf{WA}}$ assertion will hold. The theorem states that, assuming the abstract program doesn't fail, then (*i*) if C returns a value, then A will return the same value abstracted through the function *rx*; (*ii*) similarly, if C raises an exception, then A will also raise the same exception, abstracted through *ex*; (*iii*) finally, if A doesn't fail, then neither will C.

Our algorithm also needs to translate expressions from using concrete values to their corresponding abstract values. We use a predicate $\mathsf{abs\_var}_{\mathsf{WA}}$ with the following definition:

$$\mathsf{abs\_var}_{\mathsf{WA}}\ P\ a\ f\ a' \equiv P \longrightarrow a = f\ a'$$

This states that, assuming the precondition P is true, then the abstract value a corresponds to the concrete value a' abstracted using the function f. We use the convention that a primed version of a variable (such as a') represents the concrete version of a variable a.

Our algorithm for generating an abstracted version of the program is in the form of a set of syntax-directed rules. These translation rules can be applied in any setting, but in our context of Isabelle/HOL, we use them by (*i*) first proving the translation rules correct, and then (*ii*) using Isabelle/HOL's resolution engine to apply these rules.

[3] A challenge to solve this seemingly trivial goal was issued to three experienced verification engineers, with 10 minutes being the median time required to discharge the goal. The human effort for the *nat* version is effectively zero.

By carrying out these two steps in Isabelle/HOL, we simultaneously obtain both the abstracted program and an LCF-style proof of correctness that the abstraction is sound.

The next section describes the high-level approach of using Isabelle's resolution engine to simultaneously carry out both proofs and calculations in order to abstract our input specifications. The following sections go into further depth, filling in some of the missing details that come up when actually attempting to carry out such proofs in Isabelle/HOL.

6.3.1 High-level overview

We begin the process of abstracting a concrete program by generating a *schematic lemma* of the form:

$$\frac{\mathsf{corres}_{\mathsf{WA}}\ ?P_1\ rx\ ex\ ?A_1\ C}{\mathsf{corres}_{\mathsf{WA}}\ ?P_1\ rx\ ex\ ?A_1\ C}$$

The variable C is set to to the program we want to abstract, while *rx* and *ex* are set to an appropriate abstraction function for the return and exception values of the program respectively. The abstract program A_1 and the precondition P_1 are left unspecified (or *schematic*) and are given the notation $?A_1$ and $?P_1$ respectively. As our algorithm proceeds, these values will be incrementally instantiated.

For our midpoint example above, for instance, we start with the tautology:

$$\frac{\mathsf{corres}_{\mathsf{WA}}\ ?P_1\ \mathsf{unat}\ \mathsf{id}\ ?A_1\ (\mathsf{return}_{\mathsf{E}}\ ((l\ +_{\mathsf{w}}\ r)\ \mathsf{div}_{\mathsf{w}}\ 2))}{\mathsf{corres}_{\mathsf{WA}}\ ?P_1\ \mathsf{unat}\ \mathsf{id}\ ?A_1\ (\mathsf{return}_{\mathsf{E}}\ ((l\ +_{\mathsf{w}}\ r)\ \mathsf{div}_{\mathsf{w}}\ 2))}$$

Our goal is to discharge the assumption, leaving only the conclusion. We find a rule from our ruleset that pattern-matches the concrete program. Table 6.2, Table 6.3, and Table 6.4 show a representative sample of the word abstraction rules used. In this example, we wish to abstract the $\mathsf{return}_{\mathsf{E}}$ expression in our concrete program, using the rule AbsStmtReturn. This instantiates A_1 to $\mathsf{return}_{\mathsf{E}}\ ?A_2$, where $?A_2$ is a new schematic variable. Similarly, $?P_1$ is instantiated to $(\lambda s.\ ?P_2)$:

$$\frac{\mathsf{abs\_var}_{\mathsf{WA}}\ ?P_2\ ?A_2\ \mathsf{unat}\ ((l\ +_{\mathsf{w}}\ r)\ \mathsf{div}_{\mathsf{w}}\ 2)}{\mathsf{corres}_{\mathsf{WA}}\ (\lambda s.\ ?P_2)\ \mathsf{unat}\ \mathsf{id}\ (\mathsf{return}_{\mathsf{E}}\ ?A_2)\ (\mathsf{return}_{\mathsf{E}}\ ((l\ +_{\mathsf{w}}\ r)\ \mathsf{div}_{\mathsf{w}}\ 2))}$$

After applying the rule, we are now no longer abstracting program statements, but an expression inside the program, so we are now required to solve a predicate $\mathsf{abs\_var}_{\mathsf{WA}}$. We again find a rule that matches this new proposition; in this case, AbsExprDiv:

$$\frac{\begin{array}{c}\mathsf{abs\_var}_{\mathsf{WA}}\ ?P_3\ ?A_3\ \mathsf{unat}\ (l\ +_{\mathsf{w}}\ r)\\ \mathsf{abs\_var}_{\mathsf{WA}}\ ?P_4\ ?A_4\ \mathsf{unat}\ 2\end{array}}{\mathsf{corres}_{\mathsf{WA}}\ (\lambda s.\ ?P_3\ \wedge\ ?P_4)\ \mathsf{unat}\ \mathsf{id}\ (\mathsf{return}_{\mathsf{E}}\ (?A_3\ \mathsf{div}\ ?A_4))\ (\mathsf{return}_{\mathsf{E}}\ ((l\ +_{\mathsf{w}}\ r)\ \mathsf{div}_{\mathsf{w}}\ 2))}$$

Applying the rule leaves us with two new assumptions to discharge. Solving the first will instantiate $?A_3$, the left-hand side of the division, while discharging the second will

Statements

$$\frac{\forall s.\ \mathsf{abs\_var}_{\mathsf{WA}}\ (P\ s)\ a\ rx\ a'}{\mathsf{corres}_{\mathsf{WA}}\ P\ rx\ ex\ (\mathsf{return}_{\mathsf{E}}\ a)\ (\mathsf{return}_{\mathsf{E}}\ a')}\ \textsc{AbsStmtReturn}$$

$$\frac{\forall s.\ \mathsf{abs\_var}_{\mathsf{WA}}\ (P\ s)\ (a\ s)\ rx\ (a'\ s)}{\mathsf{corres}_{\mathsf{WA}}\ P\ rx\ ex\ (\mathsf{gets}_{\mathsf{E}}\ a)\ (\mathsf{gets}_{\mathsf{E}}\ a')}\ \textsc{AbsStmtGets}$$

$$\frac{\forall s.\ \mathsf{abs\_var}_{\mathsf{WA}}\ (P\ s)\ (a\ s)\ \mathsf{id}\ (a'\ s)}{\mathsf{corres}_{\mathsf{WA}}\ P\ rx\ ex\ (\mathsf{modify}_{\mathsf{E}}\ a)\ (\mathsf{modify}_{\mathsf{E}}\ a')}\ \textsc{AbsStmtModify}$$

$$\frac{\forall s.\ \mathsf{abs\_var}_{\mathsf{WA}}\ (P\ s)\ (G\ s)\ \mathsf{id}\ (G'\ s)}{\mathsf{corres}_{\mathsf{WA}}\ (\lambda\_.\ \mathsf{True})\ rx\ ex\ (\mathsf{guard}_{\mathsf{E}}\ (\lambda s.\ P\ s \wedge G\ s))\ (\mathsf{guard}_{\mathsf{E}}\ G')}\ \textsc{AbsStmtGuard}$$

$$\frac{\begin{array}{c}\mathsf{introduce\_typ\_abs\_fn}\ rx_1 \\ \mathsf{corres}_{\mathsf{WA}}\ P\ rx_1\ ex\ L\ L' \\ \forall a\ a'.\ \mathsf{abs\_expr}_{\mathsf{WA}}\ a\ rx_1\ a' \longrightarrow \mathsf{corres}_{\mathsf{WA}}\ (Q\ a)\ rx_2\ ex\ (R\ a)\ (R'\ a')\end{array}}{\begin{array}{l}\mathsf{corres}_{\mathsf{WA}}\ P\ rx_2\ ex \\ \quad (\mathbf{do}\ v \leftarrow L;\ \mathsf{guard}_{\mathsf{E}}\ (Q\ v);\ R\ v\ \mathbf{od})\ (L' \gg\!=_{\mathsf{E}} R')\end{array}}\ \textsc{AbsStmtBind}$$

$$\frac{\begin{array}{c}\mathsf{introduce\_typ\_abs\_fn}\ ex_1 \\ \mathsf{corres}_{\mathsf{WA}}\ P\ rx\ ex_1\ L\ L' \\ \forall a\ a'.\ \mathsf{abs\_expr}_{\mathsf{WA}}\ a\ ex_1\ a' \longrightarrow \mathsf{corres}_{\mathsf{WA}}\ (Q\ a)\ rx\ ex_2\ (R\ a)\ (R'\ a')\end{array}}{\mathsf{corres}_{\mathsf{WA}}\ P\ rx\ ex_2\ (\mathsf{catch}_{\mathsf{E}}\ L\ (\lambda v.\ \mathbf{do}\ \mathsf{guard}_{\mathsf{E}}\ (Q\ v);\ R\ v\ \mathbf{od}))\ (\mathsf{catch}_{\mathsf{E}}\ L'\ R')}\ \textsc{AbsStmtCatch}$$

$$\frac{\begin{array}{c}\mathsf{corres}_{\mathsf{WA}}\ P_L\ rx\ ex\ L\ L' \qquad \mathsf{corres}_{\mathsf{WA}}\ P_R\ rx\ ex\ R\ R' \\ \forall s.\ \mathsf{abs\_var}_{\mathsf{WA}}\ (P_C\ s)\ (C\ s)\ \mathsf{id}\ (C'\ s)\end{array}}{\begin{array}{l}\mathsf{corres}_{\mathsf{WA}}\ P_C\ rx\ ex \\ \quad (\mathsf{condition}_{\mathsf{E}}\ C\ (\mathbf{do}\ \mathsf{guard}_{\mathsf{E}}\ P_L;\ L\ \mathbf{od})\ (\mathbf{do}\ \mathsf{guard}_{\mathsf{E}}\ P_R;\ R\ \mathbf{od})) \\ \quad (\mathsf{condition}_{\mathsf{E}}\ C'\ L'\ R')\end{array}}\ \textsc{AbsStmtCond}$$

$$\frac{\begin{array}{c}\mathsf{abs\_var}_{\mathsf{WA}}\ P_r\ r\ rx\ r' \\ \forall r\ r'\ s.\ \mathsf{abs\_expr}_{\mathsf{WA}}\ r\ rx\ r' \longrightarrow \mathsf{abs\_var}_{\mathsf{WA}}\ (P_c\ r\ s)\ (C\ r\ s)\ \mathsf{id}\ (C'\ r'\ s) \\ \forall r\ r'.\ \mathsf{abs\_expr}_{\mathsf{WA}}\ r\ rx\ r' \longrightarrow \mathsf{corres}_{\mathsf{WA}}\ (P_b\ r)\ rx\ ex\ (B\ r)\ (B'\ r')\end{array}}{\begin{array}{l}\mathsf{corres}_{\mathsf{WA}}\ (\lambda s.\ P_r\ r\ s \wedge P_c\ r\ s)\ rx\ ex \\ \quad (\mathsf{whileLoop}_{\mathsf{E}}\ C \\ \qquad (\lambda r.\ \mathbf{do}\ \mathsf{guard}_{\mathsf{E}}\ (P_b\ r);\ r' \leftarrow B\ r;\ \mathsf{guard}_{\mathsf{E}}\ (P_c\ r');\ \mathsf{return}_{\mathsf{E}}\ r'\ \mathbf{od})\ r) \\ \quad (\mathsf{whileLoop}_{\mathsf{E}}\ C'\ B'\ r')\end{array}}\ \textsc{AbsStmtWhile}$$

Table 6.2: A selection of word abstraction rules for decomposing statements and expressions.

Signed arithmetic expressions

$$\frac{\text{abs\_var}_{\text{WA}}\ P\ a\ \text{sint}\ a' \qquad \text{abs\_var}_{\text{WA}}\ Q\ b\ \text{sint}\ b'}{\text{abs\_var}_{\text{WA}}\ (P \wedge Q)\ (a = b)\ \text{id}\ (a' = b')}\ \textsc{AbsExprEq}$$

$$\frac{\text{abs\_var}_{\text{WA}}\ P\ a\ \text{sint}\ a' \qquad \text{abs\_var}_{\text{WA}}\ Q\ b\ \text{sint}\ b'}{\text{abs\_var}_{\text{WA}}\ (P \wedge Q)\ (a < b)\ \text{id}\ (a' <_{\text{s}} b')}\ \textsc{AbsExprLe}$$

$$\frac{\text{introduce\_typ\_abs\_fn sint} \qquad \text{abs\_var}_{\text{WA}}\ P\ a\ \text{sint}\ a' \qquad \text{abs\_var}_{\text{WA}}\ Q\ b\ \text{sint}\ b'}{\begin{array}{c}\text{abs\_var}_{\text{WA}}\ (P \wedge Q \wedge \text{INT\_MIN} \leq a + b \\ \wedge\ a + b \leq \text{INT\_MAX}) \\ (a + b)\ \text{sint}\ (a' +_{\text{s}} b')\end{array}}\ \textsc{AbsExprSum}$$

$$\frac{\text{introduce\_typ\_abs\_fn sint} \qquad \text{abs\_var}_{\text{WA}}\ P\ a\ \text{sint}\ a' \qquad \text{abs\_var}_{\text{WA}}\ Q\ b\ \text{sint}\ b'}{\begin{array}{c}\text{abs\_var}_{\text{WA}}\ (P \wedge Q \wedge \text{INT\_MIN} \leq a\ \text{sdiv}\ b \\ \wedge\ a\ \text{sdiv}\ b \leq \text{INT\_MAX}) \\ (a\ \text{sdiv}\ b)\ \text{sint}\ (a'\ \text{div}_{\text{s}}\ b')\end{array}}\ \textsc{AbsExprDiv}$$

$$\frac{\text{abs\_var}_{\text{WA}}\ P\ a\ \text{sint}\ a'}{\text{abs\_var}_{\text{WA}}\ (P \wedge -\ a \leq \text{INT\_MAX})\ (-\ a)\ \text{sint}\ (-\ a')}\ \textsc{AbsExprNegate}$$

Unsigned arithmetic expressions

$$\frac{\text{introduce\_typ\_abs\_fn unat} \qquad \text{abs\_var}_{\text{WA}}\ P\ a\ \text{unat}\ a' \qquad \text{abs\_var}_{\text{WA}}\ Q\ b\ \text{unat}\ b'}{\text{abs\_var}_{\text{WA}}\ (P \wedge Q)\ (a = b)\ \text{id}\ (a' = b')}\ \textsc{AbsExprUEq}$$

$$\frac{\text{introduce\_typ\_abs\_fn unat} \qquad \text{abs\_var}_{\text{WA}}\ P\ a\ \text{unat}\ a' \qquad \text{abs\_var}_{\text{WA}}\ Q\ b\ \text{unat}\ b'}{\text{abs\_var}_{\text{WA}}\ (P \wedge Q)\ (a \leq b)\ \text{id}\ (a' \leq_{\text{w}} b')}\ \textsc{AbsExprULt}$$

$$\frac{\text{abs\_var}_{\text{WA}}\ P\ a\ \text{unat}\ a' \qquad \text{abs\_var}_{\text{WA}}\ Q\ b\ \text{unat}\ b'}{\text{abs\_var}_{\text{WA}}\ (P \wedge Q \wedge a + b \leq \text{UINT\_MAX})\ (a + b)\ \text{unat}\ (a' +_{\text{w}} b')}\ \textsc{AbsExprUAdd}$$

$$\frac{\text{abs\_var}_{\text{WA}}\ P\ a\ \text{unat}\ a' \qquad \text{abs\_var}_{\text{WA}}\ Q\ b\ \text{unat}\ b'}{\text{abs\_var}_{\text{WA}}\ (P \wedge Q)\ (a\ \text{div}\ b)\ \text{unat}\ (a'\ \text{div}_{\text{w}}\ b')}\ \textsc{AbsExprUDiv}$$

Table 6.3: A selection of word abstraction rules for abstracting expression from signed words to integers and unsigned words to naturals.

Expression decomposition

$$\frac{}{\mathsf{abs\_var}_{\mathsf{WA}}\ \mathsf{True}\ (f\ b)\ f\ b}\ \textsc{AbsExprTrivial}$$

$$\frac{\mathsf{abs\_expr}_{\mathsf{WA}}\ a\ f\ a'}{\mathsf{abs\_var}_{\mathsf{WA}}\ \mathsf{True}\ a\ f\ a'}\ \textsc{AbsExprFromAbsVar} \qquad \frac{\forall v.\ \mathsf{abs\_var}_{\mathsf{WA}}\ P\ (a\ v)\ \mathsf{id}\ (a'\ v)}{\mathsf{abs\_var}_{\mathsf{WA}}\ P\ a\ \mathsf{id}\ a'}\ \textsc{AbsExprLambda}$$

$$\frac{\mathsf{abs\_var}_{\mathsf{WA}}\ Q\ b\ \mathsf{id}\ b' \qquad \mathsf{abs\_var}_{\mathsf{WA}}\ P\ a\ \mathsf{id}\ a'}{\mathsf{abs\_var}_{\mathsf{WA}}\ (P \wedge Q)\ (f\ (a\ b))\ f\ (a'\ \$\ b')}\ \textsc{AbsExprFunApp}$$

Table 6.4: A selection of word abstraction rules for decomposing statements and expressions.

instantiate $?A_4$, the right-hand side. We continue this process of discharging assumptions and creating new ones, using the rules AbsExprSum and AbsExprTrivial, until we have no remaining assumptions and are left with just the conclusion:

$$\begin{aligned}&\mathsf{corres}_{\mathsf{WA}}\ (\lambda s.\ \mathsf{unat}\ l + \mathsf{unat}\ r \leq \mathsf{UINT\_MAX})\ \mathsf{unat}\ \mathsf{id}\\ &\quad(\mathsf{return}_{\mathsf{E}}\ ((\mathsf{unat}\ l + \mathsf{unat}\ r)\ \mathsf{div}\ 2))\\ &\quad(\mathsf{return}_{\mathsf{E}}\ ((l +_{\mathsf{w}} r)\ \mathsf{div}_{\mathsf{w}}\ 2))\end{aligned}$$

The values unat l and unat r correspond to the abstract versions of our concrete program's input parameters. To convert this theorem into an abstract function, we replace unat l and unat r with fresh variables. Additionally, the theorem only holds under the precondition that unat l + unat $r \leq$ UINT_MAX; we prepend a $\mathsf{guard}_{\mathsf{E}}$ statement ensuring that this holds. The generated abstraction thus becomes:

$$\begin{aligned}&\textbf{do}\ \mathsf{guard}_{\mathsf{E}}\ (\lambda s.\ l + r \leq \mathsf{UINT\_MAX});\\ &\quad\mathsf{return}_{\mathsf{E}}\ (l + r\ \mathsf{div}\ 2)\\ &\textbf{od}\end{aligned}$$

where the fresh variables l and r are unbounded integers.

The next sections look into some of the low-level details of this approach, and how they are implemented in Isabelle/HOL.

6.3.2 Implementation in Isabelle/HOL

The previous section gave a high-level description of the resolution-based approach AutoCorres uses to abstract programs. One problem left unresolved is how we establish a formal connection between low-level concrete variables and their abstract equivalents. For example, if we wish to abstract the word-typed expression $a' +_{\mathsf{w}} b' = c'$ to its integer-typed equivalent $a + b = c$, we need to have formal connections between a' and a; b' and b; and so on. Furthermore, during the resolution-based abstraction algorithm, we need to keep track of such existing connections, as well as introduce new abstract variables as we encounter new concrete variables.

We begin to address this by using a predicate $\text{abs\_expr}_{\text{WA}}$, defined as follows:

$$\text{abs\_expr}_{\text{WA}}\ a\ f\ a' \equiv a = f\ a'$$

This predicate states that the value a is the abstract version of the variable a', using the abstraction function f.

For each variable in the input concrete program, we generate a corresponding abstract variable and add a $\text{abs\_expr}_{\text{WA}}$ assumption to the current goal. For example, the premise of the midpoint example above would start as[4]

$$\frac{\text{abs\_expr}_{\text{WA}}\ l\ \text{unat}\ l' \qquad \text{abs\_expr}_{\text{WA}}\ r\ \text{unat}\ r'}{\text{corres}_{\text{WA}}\ ?P_1\ \text{unat}\ \text{id}\ ?A_1\ (\text{return}_{\text{E}}\ ((l' +_{\text{w}} r')\ \text{div}_{\text{w}}\ 2))}$$

Our aim is to reduce the goal to True by breaking it into smaller, simpler parts; during this process, the schematic variables P_1 and A_1 will be instantiated. The two $\text{abs\_expr}_{\text{WA}}$ terms can be used to discharge side conditions in the rules we apply to the goal. For example, the rule AbsExprFromAbsVar

$$\frac{\text{abs\_expr}_{\text{WA}}\ a\ f\ a'}{\text{abs\_var}_{\text{WA}}\ \text{True}\ a\ f\ a'}$$

states that an expression consisting only of a single variable can be translated into the abstract version of that variable.

During the process of abstraction, we will come across new concrete variables that need to be abstracted. In this case, new $\text{abs\_expr}_{\text{WA}}$ predicates can be introduced into our set of assumptions. The rule AbsStmtBind is an example where this occurs:

$$\frac{\begin{array}{c}\text{introduce\_typ\_abs\_fn}\ rx_1 \\ \text{corres}_{\text{WA}}\ P\ rx_1\ ex\ L\ L' \\ \forall a\ a'.\ \text{abs\_expr}_{\text{WA}}\ a\ rx_1\ a' \longrightarrow \text{corres}_{\text{WA}}\ (Q\ a)\ rx_2\ ex\ (R\ a)\ (R'\ a')\end{array}}{\begin{array}{l}\text{corres}_{\text{WA}}\ P\ rx_2\ ex \\ \quad (\textbf{do}\ v \leftarrow L;\ \text{guard}_{\text{E}}\ (Q\ v);\ R\ v\ \textbf{od})\ (L' \gg\!=_{\text{E}} R')\end{array}}$$

The rule above abstracts the monadic bind operator by first abstracting the left-hand side (i.e., the first statement), then abstracting the right-hand side (i.e., the second statement), and then finally combining the results. Any precondition generated while abstracting the right-hand side is checked by the guard_{E} statement emitted on the abstract side.[5]

The left-hand side of the bind operator introduces a new bound variable that is passed into the right-hand side. When we abstract the right-hand side, we introduce a new variable a corresponding to the concrete variable a', and also introduce a new assumption $\text{abs\_expr}_{\text{WA}}\ a\ rx_1\ a'$ that can be used when abstracting the right-hand side.

[4] We no longer show both the premise and the under-construction conclusion as we did for our earlier examples, but simply the premise. The under-construction conclusion remains in the background however, tracked by Isabelle/HOL.

[5] Frequently, this will simply be $\text{guard}_{\text{E}}\ (\lambda s.\ \text{True})$, and will be optimised away in a later phase of AutoCorres.

The type chosen to abstract a' into (and hence the function chosen for rx_t) is determined by the predicate introduce_typ_abs_fn. This predicate has a trivial definition:

$$\text{introduce\_typ\_abs\_fn } f \equiv \text{True}$$

but its presence allows us to control what variables are abstracted into. For example, if we only want to abstract *word32* types and not their signed equivalent, we can include the rule introduce_typ_abs_fn unat in the ruleset used for abstraction, where unat converts *word32* types to *nat*. If we also wanted to abstract signed values, then we also include the rule introduce_typ_abs_fn sint where sint converts *sword32* types to *int*.

6.4 Word abstraction examples

This section provides some simple examples of word abstraction. We start with a few trivial examples demonstrating how word abstraction works in practice on smaller expressions. We move on to a large example of a primality testing function, and provide a brief comparison of performing the proof with and without using word abstraction.

6.4.1 Maximum of two integers

Our first example is a simple `max` function, which takes two input integers and returns the maximum of the two. Figure 6.1 shows the C source code to the function, as well as the intermediate states of AutoCorres before word abstraction has taken place (max_{L_2}) and after word abstraction (max_{WA}). While the two monadic versions look almost identical, the significant differences exist in the types of the two functions: max_{L_2} has variables and a return type of *sword32*, while max_{WA} has variables and a return type of *int*.

After the remaining stages of AutoCorres (including type-strengthening previously described in Section 5.3), the final version of our `max` program is as follows:

$$\begin{aligned}&\text{max}'\ a\ b \equiv \\ &\qquad \text{if } a \leq b \text{ then } b \text{ else } a\end{aligned}$$

This final definition is identical (in both type and body) to the built-in max definition of Isabelle.

6.4.2 Absolute value

Our second example is a simple *absolute value* function written in C, shown in Figure 6.2. While this function is seemingly trivial, a source of trouble arises from the fact that the signed integer value -2^{31} doesn't have an equivalent positive value; that is, $-(-2^{31}) \neq 2^{31}$, as the value 2^{31} cannot be represented as a 32-bit signed value. The C standard does not attempt to specify what a C program will do if it attempts to

```
int max(int a, int b) {
    if (a <= b)
        return b;
    return a;
}
```

$$\mathsf{max}_{\mathsf{L}_2}\ a\ b \equiv \mathsf{condition}_{\mathsf{E}}\ (\lambda s.\ a \leq_{\mathsf{s}} b)\ (\mathsf{return}_{\mathsf{E}}\ b)\ (\mathsf{return}_{\mathsf{E}}\ a)$$

$$\mathsf{max}_{\mathsf{WA}}\ a\ b \equiv \mathsf{condition}_{\mathsf{E}}\ (\lambda s.\ a \leq b)\ (\mathsf{return}_{\mathsf{E}}\ b)\ (\mathsf{return}_{\mathsf{E}}\ a)$$

Figure 6.1: An implementation of a function `max` in ANSI C, returning the maximum of the two input parameters. Also shown are the intermediate states of AutoCorres before ($\mathsf{max}_{\mathsf{L}_2}$) and after ($\mathsf{max}_{\mathsf{WA}}$) word abstraction of the max function.

carry out such an operation, leaving it as an undefined behaviour. To ensure that such undefined behaviour does not occur, Norrish's C-to-Isabelle parser places the guard $-\ \mathsf{sint}\ x \leq \mathsf{INT\_MAX}$ prior to the unary minus operation.

When AutoCorres abstracts the expression $-x$ from the type *sword32* to *int* using the rule AbsExprNegate, a new guard must be emitted for the conversion to be sound. This results in the body of the conditional have the following intermediate value:

$$\begin{array}{l}\mathsf{condition}_{\mathsf{E}}\ (\lambda s.\ x < 0)\\ \quad (\textbf{do}\ \mathsf{guard}_{\mathsf{E}}\ (\lambda s.\ -\ x \leq \mathsf{INT\_MAX});\\ \quad\quad \mathsf{guard}_{\mathsf{E}}\ (\lambda s.\ -\ x \leq \mathsf{INT\_MAX});\\ \quad\quad \mathsf{return}_{\mathsf{E}}\ (-\ x)\\ \quad \textbf{od})\\ \quad (\mathsf{return}_{\mathsf{E}}\ x)\end{array}$$

Here, the first $\mathsf{guard}_{\mathsf{E}}$ statement is the original translated from $\mathsf{abs}_{\mathsf{L}_2}$, while the second is generated when translating the expression $-x$. The flow-sensitive optimisations described in Chapter 5 remove the redundant guard, leaving the user with just a single guard, similar to the input program.[6]

6.4.3 Primality testing

In this example, we use AutoCorres' word abstraction to verify the correctness of a C function `is_prime`. The function takes a single unsigned integer and determines if it is a prime number or not. We start by proving a simple $O(n)$-time implementation of the algorithm, and then extend the program and proof to work using a more efficient $O(\sqrt{n})$ algorithm.

Our initial linear time C implementation `is_prime_linear` is shown in Figure 6.3. The output of AutoCorres with unsigned word abstraction enabled is shown in Figure 6.4. Word abstraction transforms the *word32* types into natural numbers. The only

[6] In fact, *every* expression translated by word abstraction results in a guard being generated; most of them, however, are simply $\mathsf{guard}_{\mathsf{E}}\ (\lambda s.\ \mathsf{True})$, and thus are easily removed by the peephole and flow-sensitive optimisations discussed in Chapter 5.

```
int abs(int x) {
    if (x < 0)
        return -x;
    return x;
}
```

$\text{abs}_{L_2}\ x \equiv$
$\quad \text{condition}_E\ (\lambda s.\ x <_s 0)$
$\quad\quad (\textbf{do}\ \text{guard}_E\ (\lambda s.\ -\ \text{sint}\ x \leq \text{INT\_MAX});$
$\quad\quad\quad \text{return}_E\ (-\ x)$
$\quad\quad \textbf{od})$
$\quad\quad (\text{return}_E\ x)$

$\text{abs}_{WA}\ x \equiv$
$\quad \text{condition}_E\ (\lambda s.\ x < 0)$
$\quad\quad (\textbf{do}\ \text{guard}_E\ (\lambda s.\ -\ x \leq \text{INT\_MAX});$
$\quad\quad\quad \text{return}_E\ (-\ x)$
$\quad\quad \textbf{od})$
$\quad\quad (\text{return}_E\ x)$

Figure 6.2: An implementation of the absolute value function `abs` in ANSI C. Also shown are the intermediate states of AutoCorres before (abs_{L_2}) and after (abs_{WA}) word abstraction of the `abs` function.

```
/* Determine if the input number 'n' is prime. */
unsigned is_prime_linear(unsigned n)
{
    /* Numbers less than 2 are not prime. */
    if (n < 2)
        return 0;

    /* Find the first non-trivial factor of 'n'. */
    for (unsigned i = 2; i < n; i++) {
        if (n % i == 0)
            return 0;
    }

    /* No factors. */
    return 1;
}
```

Figure 6.3: A linear time implementation of a prime checking function in C.

change in the program's structure is a single new guard introduced above the expression $i +_w 1$ at the end of the loop body. This guard ensures that the loop counter `i` does not overflow, which would make AutoCorres' abstraction from *word32*'s to natural numbers unsound.

Verifying `is_prime_linear` with word abstraction

To verify the correctness of `is_prime_linear`, we wish to prove the property:

$$\{\lambda s.\ n \leq \text{UINT\_MAX}\}\ \text{is\_prime\_linear}'\ n\ \{\lambda r\ s.\ (r \neq 0) = \text{prime}\ n\}$$

That is, assuming that we pass is_prime_linear′ an input integer n no larger than UINT_MAX, then the function will return a non-zero value if and only if n is prime, where the function prime is the function from Isabelle/HOL's standard library.

The proof of correctness takes place in the following steps:

Lemmas for $\mathbf{O}(n)$-time `is_prime_linear`

$$\text{partial\_prime } p\ (n+1) = (\text{partial\_prime } p\ n \wedge (1 < n \wedge n + 1 < p \longrightarrow \neg\ n \text{ dvd } p))$$
PartialPrimeSuc

$$\text{prime } (a * b) = (a = 1 \wedge \text{prime } b \vee \text{prime } a \wedge b = 1)$$
PrimeOfProduct

$$(n \bmod i \neq 0) = (\neg\ i \text{ dvd } n)$$
ModZeroToDvd

$$(x \text{ dvd } x + 1) = (x = 1)$$
DivideSelfPlusOne

Lemmas for $\mathbf{O}(\sqrt{n})$-time `is_prime`

$$\frac{p < n * n}{\text{partial\_prime } p\ n = \text{prime } p}$$
PartialPrimeSqr

$$\text{UINT\_MAX} = \text{SQRT\_UINT\_MAX} * \text{SQRT\_UINT\_MAX} - 1$$
UintMaxFactor

$$\frac{\text{prime } p}{(r \text{ dvd } p) = (r = 1 \vee r = p)}$$
PrimeDvd

$$i * i < j * j = (i < j)$$
SqrLessMono

$$\frac{b \neq 0}{a * a \leq b * b - 1 = (a < b)}$$
PartialPrimeSqr

Table 6.5: Helper lemmas required for the correctness proof of `is_prime_linear` and `is_prime` to be discharged automatically. The theories all hold on the type *nat*.

```
is_prime_linear' n ≡
        condition (λs. n < 2)
            (return 0)
            (catch
                (do whileLoop_E (λi s. i < n)
                        (λi. do guard_E (λs. 0 < i);
                                when_E (n mod i = 0) (throw_E 0);
                                guard_E (λa. i + 1 ≤ UINT_MAX);
                                return_E (i + 1)
                            od) 2;
                    throw_E 1
                od)
                return)
```

Figure 6.4: The output of AutoCorres with word abstraction enabled for the `is_prime_linear` function in Figure 6.3.

1. We show that the result is correct for $n = 0$ and $n = 1$ using simple term rewriting;
2. We show that the result is correct for $n \geq 2$ by annotating the loop body with an invariant and with a termination measure; finally
3. We execute the VCG over the body of the function, resulting in four proof obligations: (*i*) that the loop invariant holds entering the loop; (*ii*) that the loop invariant holds between loop iterations; (*iii*) that the loop measure decreases each loop iteration; and (*iv*) that the loop invariant implies the calculated postcondition for the loop.

The first step, showing correctness when $n = 0$ or $n = 1$, is trivially proven by using the monadic VCG `wp` and Isabelle's simplifier.

To prove the general case, we annotate the loop with the measure $n - i$ and with the loop invariant:

$$1 < i \wedge 1 < n \wedge i \leq n \wedge \text{partial\_prime } n\ i$$

Here, the predicate partial_prime is defined as:

$$\text{partial\_prime } n\ k \equiv 1 < n \wedge (\forall i \in \{2..{<}\text{min } n\ k\}.\ \neg\ i \text{ dvd } n)$$

This states that the number n has no non-trivial factors less than k. The notation $\{a..{<}b\}$ refers to the set of integers between a and b, including a but excluding b.

We can use partial_prime to show that a number is prime: once its second argument k is "big enough", we know that n has no non-trivial factors, and hence is prime. An easy choice for k that arises from the definition of prime is simply to choose n:[7]

$$\frac{n \leq k}{\text{partial\_prime } n\ k = \text{prime } n}$$

[7] We tighten the bound of k below when improving our algorithm.

With the introduction of some helper lemmas describing basic facts about prime numbers and the partial_prime predicate, all of the goals above can be discharged automatically using Isabelle's built-in `auto` tactic. The helper lemmas required are shown in Table 6.5.

Verifying `is_prime_linear` without word abstraction

How much effort does word abstraction save on such a proof? Attempting to carry out the proof directly on the *word32* type, we very quickly run into problems. For instance, if we attempt to use a definition of prime for the *word32* type similar to that of the natural type:

$$\text{prime } p = (1 < p \wedge (\forall m.\ m \text{ dvd } p \longrightarrow m = 1 \vee m = p))$$

which uses the corresponding definition of dvd:

$$b \text{ dvd } a = (\exists k.\ a = b * k)$$

we can then prove the rather inconvenient fact that primes do not exist:

$$\{x.\ \text{prime } x\} = \emptyset$$

What went wrong? The proof above stems from our use of the *word32* type and our definition of dvd above. According to this definition, the number 3 divides everything (i.e., $\forall n.\ 3 \text{ dvd } n$). This is because

$$3 *_w \text{0xAAAAAAAB} = 1$$

due to overflow; and hence every number n can be factored into the expression of the form

$$(3 *_w n) *_w \text{0xAAAAAAAB}$$

Hence, no number is prime.[8]

To work around these issues, we could attempt to redefine dvd on the *word32* type to a definition that ignores results involving overflow. We would then need to derive for our new dvd function the large library of results already available for the original dvd in the Isabelle library. Even so, our troubles would not be over. For example, even using our new dvd function, the rule DivideSelfPlusOne:

$$x \text{ dvd } x + 1 = (x = 1)$$

is false when $x = 2^{32} -_w 1$, as $x +_w 1 = 0$, and $2^{32} -_w 1$ dvd 0.

[8] In fact, every number in the *word32* type can be factored in a large number of ways; every odd number n has an inverse modulo 2^{32}, termed n^{-1}. Every such inverse can be used to factor every other number m such that $m = (m \times n) \times n^{-1}$.

Attempting to reason directly with the *word32* type is hard work.[9] An alternative route would be to lift all our reasoning to the naturals. For instance, our loop invariant would become:

$$1 < \text{unat } i \wedge 1 < \text{unat } n \wedge \text{unat } i \leq \text{unat } n \wedge \text{partial\_prime } (\text{unat } n)\ (\text{unat } i)$$

Performing our high-level reasoning over the natural numbers avoids the pathological overflow cases described above, but still forces us to deal with tens of mundane overflow cases, such as when we want to show unat $(i +_w 1) = \text{unat } i + 1$ (which is only true if $i + 1 < 2^{32}$). Such proof obligations can be discharged by having the appropriate invariants so that we can show that $i + 1$ does not overflow.

Once we lift all our reasoning from the *word32* to the naturals, we are effectively just performing the word abstraction process by hand: automatic word abstraction saves the user the burden of having to do it herself.

Improving the algorithm

The algorithm used by `is_prime_linear` requires $O(n)$ time to determine if an input n is prime. We can easily improve the run-time to $O(\sqrt{n})$ by observing that if n has a non-trivial factor a, then it will have a factor $b \leq \sqrt{n}$. We can formalise this by improving the bound on the partial_prime theorem above:

$$\frac{p < n * n}{\text{partial\_prime } p\ n = \text{prime } p}$$

We prove this by showing that every composite n has two non-trivial factors a_1 and a_2, where $a_1 \times a_2 = n$. Without loss of generality, assume $a_1 \leq a_2$. Then, by monotonicity of multiplication, $a_1 \times a_1 \leq a_1 \times a_2 = n$. Thus, if there is no number n such that $n \times n \leq p$ divides p, then p must be a prime.

With this in mind, we can change the main loop of our program from:

```
/* Find the first non-trivial factor of 'n'. */
for (unsigned i = 2; i < n; i++) {
    if (n % i == 0)
        return 0;
}
```

to:

```
/* Find the first non-trivial factor of 'n' less than sqrt(n). */
for (unsigned i = 2; i < SQRT_UINT_MAX && i * i <= n; i++) {
    if (n % i == 0)
        return 0;
}
```

where `SQRT_UINT_MAX` is 65536.

[9] This example was not specifically cooked up to trip up the word-based proof—our experience shows these issues crop up regularly for any non-trivial arithmetic results that are required in the process of program verification.

```
is_prime' n ≡
        condition (λs. n < 2)
            (return 0)
            (catch
                (do whileLoop_E (λi s. i < SQRT_UINT_MAX ∧ i * i ≤ n)
                        (λi. do guard_E (λs. 0 < i);
                                when_E (n mod i = 0) (throw_E 0);
                                i ← return_E (i + 1);
                                guard_E (λs. i < SQRT_UINT_MAX ⟶ i * i ≤ UINT_MAX);
                                return_E i
                            od) 2;
                    throw_E 1
                od)
                return)
```

Figure 6.5: The output of AutoCorres from an $O(\sqrt{n})$-time implementation of the `is_prime` function.

The primary difference between the two loops is the exit condition, which has been modified from `i < n` (i.e., try all potential factors up to n) to `i * i <= n` (i.e., try all potential factors up to and including $\sqrt{n}$).

We also need to add an additional check `i < SQRT_UINT_MAX` to ensure that `i * i` doesn't overflow; such a situation would occur when the input n satisfies $65\,535^2 < n < 65\,536^2$. Without the overflow check, the condition `i * i <= n` will never be satisfied for such an input n, causing the loop to continue until finally $n = i$ and the check `n % i == 0` succeeds. For primes in the range $65\,535^2 < n < 65\,536^2$, such as $4\,294\,836\,241$, this will cause the function to incorrectly state that the input is composite.

Running the new program through AutoCorres produces the output shown in Figure 6.5. The output is similar to the version of is_prime_linear′. The two differences are that the loop condition differs from the original, reflecting the change of the C code; and a new guard has been added at the end of the loop to ensure that the arithmetic in the loop does not overflow.[10]

To prove this adjusted version, we first update the invariant of the loop to:

$$\begin{aligned}&1 < i \wedge i \leq n \wedge i \leq \text{SQRT\_UINT\_MAX} \wedge \\ &i * i \leq \text{SQRT\_UINT\_MAX} * \text{SQRT\_UINT\_MAX} \wedge \\ &\text{partial\_prime } n\ i\end{aligned}$$

and the loop's measure to $(\lambda(r, s).\ (n + 1) * (n + 1) - r * r)$. The measure is not as tight as it could be, but this form matches the proof obligations generated by the VCG, making the verification conditions easier to discharge.

Once the loop has been annotated with an invariant and measure, we perform the same steps as previously: prove the theorem true for $n = 1$ and $n = 2$, run the VCG, and then use Isabelle's inbuilt solvers to discharge the goal. As in the previous case,

[10] Such a guard would have also been added prior to the loop, to ensure that the very first check did not overflow. The guard condition $2 < \text{SQRT\_UINT\_MAX} \longrightarrow 2 * 2 \leq \text{UINT\_MAX}$ is proven to always hold by AutoCorres' simplification phase, and hence the guard is removed.

some helper lemmas need to be added to the automated tactics; these are shown in the second half of Table 6.5. Once added, the proof obligations are solved by Isabelle's `auto` tactic without further effort.

6.5 Extending the rule set

AutoCorres has approximately 40 rules built-in to process all C statements and expressions, and uses an additional 11 for each type that needs to be abstracted (e.g., signed words and unsigned words). While typically these rules need not be modified (or even understood) by users of AutoCorres, the rule sets can be extended if the user wishes to abstract code-specific idioms that are sound at the concrete level but become unprovable after abstraction.

For instance, the function:

```
int sum_overflows(unsigned a, unsigned b) {
    return a + b < a;
}
```

determines if the unsigned addition of `a` and `b` overflows; this is done by performing the (potentially overflowing) addition, and then comparing the result to ensure it is larger than the inputs.

If we carry out AutoCorres' unsigned word abstraction, the result of the statement after simplifications occur is:

$$\begin{array}{l}\textbf{do}\ \mathsf{guard}\ (\lambda s.\ a + b \leq \mathsf{UINT\_MAX});\\ \quad \mathsf{return}\ 0\\ \textbf{od}\end{array}$$

After abstraction, instead of the expression being a check for overflow, the user must now *prove* that the overflow cannot occurs. As a further insult, after the abstraction has taken place AutoCorres' simplification routines determine that the expression $a + b < a$ over the naturals is always false, so simplifies the entire expression to 0 (i.e., `false`). The test has become useless.

By extending the word abstraction ruleset with the following rule, the user can still continue to use unsigned word abstraction:

$$\frac{\mathsf{abs\_var}_{\mathsf{WA}}\ P\ x\ \mathsf{unat}\ x' \qquad \mathsf{abs\_var}_{\mathsf{WA}}\ Q\ y\ \mathsf{unat}\ y'}{\mathsf{abs\_var}_{\mathsf{WA}}\ (P \wedge Q)\ (\mathsf{UINT\_MAX} < x + y)\ \mathsf{id}\ (x' +_{\mathsf{w}} y' <_{\mathsf{w}} x')}$$

This rule states that expressions of the form $a + b < a$ should be abstracted into the overflow check $\mathsf{UINT\_MAX} < x + y$. After running word abstraction with this added rule, the function is abstracted into:

$$\mathsf{return}\ (\mathsf{if}\ \mathsf{UINT\_MAX} < a + b\ \mathsf{then}\ 1\ \mathsf{else}\ 0)$$

which captures the original intent of the concrete code.

6.6 Related work

The detailed model of word arithmetic used by the Norrish's C-to-Isabelle parser (and in turn our own work) stems from Dawson [38], which is now part of the Isabelle/HOL standard proof library. Dawson's work accurately models operations such as addition, multiplication, division and so on for unsigned words. Our own work has extended Dawson's to add support for signed operations, in particular signed division and signed modulo, which produce different results to their unsigned counterparts.

Carrying out word proofs has long been a burden for users of interactive theorem provers, and many approaches have been taken to simplify or automate them. Böhme et al. [24], for instance, exported proofs to the powerful Z3 SMT solver, replaying the proofs in HOL4 and Isabelle/HOL. Isabelle/HOL includes a tactic developed by Thomas Sewell named `word_bitwise`, that converts word proofs into a boolean circuit representation, which can often be more easily solved using Isabelle/HOL's inbuilt automation. Perhaps closest to our own is further work by Dawson [38], who developed the Isabelle/HOL tactics `unat_arith` and `uint_arith` that attempts to lift Isabelle/HOL subgoals from unsigned word arithmetic to unbounded arithmetic on integers and naturals. Like our own work, after lifting the user is required to show that intermediate calculations don't exceed the allowable range of the input types. While these different techniques attempt to either simplify or solve word proofs, our own work attempts to avoid them altogether by rewriting the input specification to operate directly on unbounded types. This has the added advantage that program preconditions, postconditions and invariants can also be written using such unbounded types.

Other C verification frameworks have different approaches to reasoning about integers. VCC [32], for instance, gives the user the option to either model word arithmetic directly as unbounded integers and naturals, with guards ensuring that results of arithmetic do not overflow; or to model word arithmetic as bit-vectors, which allow overflowing operations to be accurately modelled and verified, but at the expense of increasing the difficulty of verification by the underlying automatic provers. VCC simply emits these definitions with no formal link to the underlying machine words. This would, for example, complicate the process of creating a formal link down to the compiler assembly. Similarly, Moy's translation of C into the Frama-C framework with the Jessie plugin [72] emits unbounded integer operations with checks for overflow.

In the other direction, the verified C compiler *CompCert* [67] represents integer values as n-bit machine words, similar to that used by Norrish's C-to-Isabelle parser. This representation of machine words is more convenient for proving correspondence between the input C semantics and the generated machine code, but comes at the cost of complicating user reasoning, as discussed previously in this chapter.

In contrast to both of these approaches, AutoCorres begins with an accurate model of word arithmetic and then abstracts the specification into the unbounded model, simultaneously generating a proof of correctness. This provides a formally verified link between the low-level machine model and a model of arithmetic more convenient for users to work with.

6.7 Conclusion

This chapter has shown how monadic specifications using word-based arithmetic can be provably abstracted into higher-level specifications using unbounded integers and naturals.

Word abstraction is effective: for example, AutoCorres' output of the `max` function in Figure 6.1 *precisely* matches Isabelle's built-in definition of max on the *nat*s. AutoCorres additionally reduced the burden of proving the arithmetical `is_prime` function: instead of needing to manually lift word values into naturals, AutoCorres did this for us. More complex usages of word arithmetic invariably cause the abstracted program to also be more complex, as the user becomes obliged to prove that the arithmetic does not overflow. In our experience, however, the abstracted version tends to be far simpler to reason about than the original input program.

Chapter Summary

- The numeric types in low-level languages such as C are finite; on a 32-bit system, for instance, the range of a `signed int` is -2^{31} to $2^{31} - 1$, while the range of a `unsigned int` is 0 to $2^{32} - 1$.
- Reasoning about finite numeric types is hard for two primary reasons: (*i*) the finite types overflow, meaning that basic mathematical 'truths' such as $x < x + 1$ or $(2 \times x = 4) \longrightarrow (x = 2)$ do not hold; and (*ii*) users are required to prove that signed types do not overflow on every arithmetic operation.
- We abstract programs using signed arithmetic into programs that operate on unbounded integers. This can be done *soundly* by exploiting the fact the users must *already* prove that signed operations do not exceed their maximum range. Our program abstraction process simultaneously generates both an abstract program and a proof of its correctness.
- Similarly, we are able to abstract programs using unsigned arithmetic into programs that operate on unbounded naturals, if the user's program does not rely on unsigned overflow and they are willing to prove this.
- The word abstraction ruleset can be extended by the user to allow particular idioms used in low-level code abstracted to a suitable high-level representation.
- Word abstraction simplifies reasoning both because (*i*) reasoning on integers and naturals is more intuitive; and (*ii*) Isabelle/HOL has a large library of theorems and tactics available for reasoning about integers and natural numbers, but far fewer for finite words.

7 Heap abstraction

We have thus far neglected to think about how we can ease formal verification of C programs that need to access the system's memory. Norrish's C-to-Isabelle parser uses a byte-level model of memory which, while being a conservative choice, significantly complicates reasoning.

In this chapter we look at how we can automatically abstract C programs that use a byte-level model of memory into programs that operate on a Burstall-Bornat split heap model. Like previous chapters, our tool also generates a proof in Isabelle/HOL showing that our translation is correct.

Section 7.1 describes in detail how Norrish's C-to-Isabelle parser models system memory, and the difficulties that arise when attempting to directly reason about it. In Section 7.2 we next describe our implementation of an existing reasoning framework developed by Tuch et al. [103] used to facilitate manual reasoning about byte-level heaps. Not content with such manual reasoning, in Section 7.3 we show how we can use our reimplementation of Tuch et al.'s logic to implement automatic abstraction of C programs, translating programs to operate on a more abstract heap. The remainder of the chapter looks at additional technical details—such as dealing with C `structs` and interacting with functions that need to reason about a byte-level heap—and present some simple case studies.

This chapter is based on the published work by Greenaway et al. [50], *Don't sweat the small stuff: formal verification of C code without the pain* in PLDI 2014.

7.1 Byte-level versus typed heap reasoning

When reasoning about non-trivial C programs, a question that quickly arises is how the system's memory or *heap* should be formally modelled. One common approach

when modelling higher-level languages such as Java [87] is to represent the contents of memory locations as a datatype:

$$\textbf{datatype}\ \mathit{value} = \mathsf{Int}\ \mathit{int}\ \mid \mathsf{Float}\ \mathit{float}\ \mid \mathsf{IntPtr}\ \mathit{addr}\ \mid \cdots$$

The heap can then be modelled as a function of type $\mathit{addr} \Rightarrow \mathit{value}$ that converts a pointer address to its object value.

One difficulty presented by this heap model is that of *inter-type aliasing*. That is, if we have a pointer p to an `int` and a pointer q to a `float`, then when we write to the pointer p we must prove that $p \neq q$ in order to know that the float at q is unchanged. In larger programs, dealing with these side conditions becomes a non-trivial burden, as documented, for instance, by Burstall [23] and Bornat [18].

The solution to the problem of inter-type aliasing developed by Burstall and Bornat is to use a *split heap* model of memory. In this model, each different type has its own function that maps pointers to their logical values:

$$\begin{array}{l}\textbf{record}\ \mathit{state} = \\ \quad \mathsf{heap\_int} :: \mathit{word32} \Rightarrow \mathit{int} \\ \quad \mathsf{heap\_float} :: \mathit{word32} \Rightarrow \mathit{float} \\ \quad \mathsf{heap\_intptr} :: \mathit{word32} \Rightarrow \mathit{addr} \\ \quad \cdots\end{array}$$

The split heap model avoids the inter-type aliasing problem because updates to an `int` pointer p only modify the heap_int function, making it clear that other types contained in the other heaps are unaffected.

Both of these models of the heap are, in some sense, unsatisfactory for C programs that frequently perform byte-level accesses to memory. For instance, functions such as `memcpy` and `memcmp` access the heap in a byte-level manner, while other C programs reinterpret the same memory region as different types, such as by casting pointers, in `unions`, or after `malloc` and `free` operations.

When modelling C programs that carry out such low-level memory operations, a better model of memory may be to simply represent it as a function from addresses to individual bytes. So, on a 32-bit system, the heap would be represented by a function of type $\mathit{word32} \Rightarrow \mathit{word8}$.

On the surface, this simple approach has many benefits: it is easy to understand, faithful to how the hardware functions at a low level, and allows low-level C code that interacts with memory at a low level (such as `memcpy`, `memset`, casting pointers between types, etc.) to be reasoned about. While not perfect (for example, there is no way to represent unmapped or invalid addresses), if we are unable to reason about this simple model, we are going to struggle with anything more sophisticated.

```
void swap(unsigned *a, unsigned *b)
{
    unsigned t = *a;
    *a = *b;
    *b = t;
}
```

```
TRY
    GUARD C_Guard {|c_guard 'a|}
        't :== h_val (hrs_mem 't_hrs) 'a;;
    GUARD C_Guard {|c_guard 'a|}
        GUARD C_Guard {|c_guard 'b|}
            'globals :==
              t_hrs_'_update
                  (hrs_mem_update
                      (heap_update 'a (h_val (hrs_mem 't_hrs) 'b)));;
    GUARD C_Guard {|c_guard 'b|}
        'globals :==
          t_hrs_'_update (hrs_mem_update (heap_update 'b 't))
CATCH
    SKIP
END
```

Figure 7.1: An implementation of the `swap` function and its translation into Simpl by Norrish's C-to-Isabelle parser.

7.1.1 Norrish and Tuch byte-level heap implementation

In order to be able to support low-level reasoning about C programs, Norrish's C-to-Isabelle parser uses this byte-level model of the system heap;[1] that is, each time a C program writes to a pointer, the logical object being written is first encoded into a list of *word8*s before being written out to the heap. Similarly, each time a pointer is read from, a list of *word8*s are read from the heap. The list is then decoded back into a logical object.

Figure 7.1 shows an implementation of a `swap` function and its translation into Simpl by Norrish's C-to-Isabelle parser using this byte-level model of the heap. The syntax used by the C-to-Isabelle parser to represent reads and writes to global memory is taken from Tuch [100]. While the syntax used may appear rather opaque at first glance, it is fundamentally using the simple byte-based heap model described in the previous section.

The *globals* record generated by the C-to-Isabelle parser contains a field named t_hrs_', used to track the current state of the heap:

[1] Strictly speaking, the memory model used by Norrish's C-to-Isabelle parser is pluggable, but every implementation that the author is aware of currently uses some variant of a byte-level memory model.

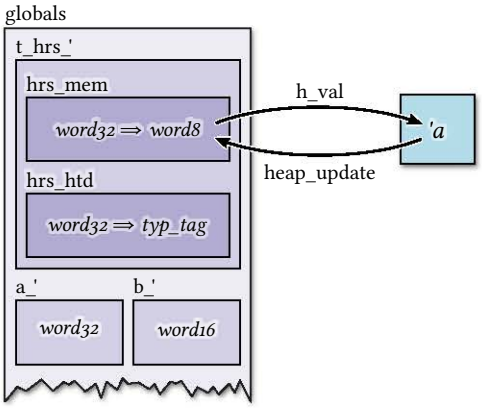


Figure 7.2: The structure of the global state used prior to heap abstraction. The globals record consists of all of the global variables used in the program (such as a_' and b_' in this figure), as well as a heap field t_hrs_′. The heap field contains two subcomponents: hrs_mem, which stores the raw bytes in memory, and hrs_htd, which stores the type tags for each byte. The functions h_val and heap_update decode and encode raw bytes into logical objects, respectively.

record *globals* =
t_hrs_′ :: (*word32* ⇒ *word8*) × (*word32* ⇒ *typ_tag*)
x_′ :: *word32*
y_′ :: *word16*
z_′ :: *word32*
...

This t_hrs_′ field consists of two parts: the first component has type *word32* ⇒ *word8*, and stores the raw bytes of the heap. It is accessed with the function hrs_mem. The second half of type *word32* ⇒ *typ_tag* is a *type tag mapping*, which stores a *type tag* for each byte in the heap, and is accessed with the function hrs_htd. We will return to the hrs_htd field later in Section 7.2. Each of the three functions t_hrs_′, hrs_mem, and hrs_htd have associated functions that update their values: the t_hrs_′ field is updated with the function t_hrs_′_update, while the two components of t_hrs_′ are updated with hrs_mem_update and hrs_htd_update, respectively.

To actually decode the bytes from the heap into logical objects, the C-to-Isabelle parser emits calls to a function h_val. The function has type

$$\text{h\_val} :: (\textit{word32} \Rightarrow \textit{word8}) \Rightarrow {'a}\ \textit{ptr} \Rightarrow {'a}$$

The expression h_val *h p* takes a heap $h :: \textit{word32} \Rightarrow \textit{word8}$ and a pointer $p :: {'a}\ \textit{ptr}$, and decodes the bytes at the location of the pointer, producing an object of type $'a$.

Conversely, the function heap_update encodes logical objects back into lists of bytes. This function has type

$$\text{heap\_update} :: {'a}\ \textit{ptr} \Rightarrow {'a} \Rightarrow (\textit{word32} \Rightarrow \textit{word8}) \Rightarrow \textit{word32} \Rightarrow \textit{word8}$$

The expression heap_update *p v h* updates the heap *h* at location $p :: {'a}\ \textit{ptr}$, writing the byte-encoded value of $v :: 'a$.

To help explain the various components of Tuch et al.'s heap model, Figure 7.2 depicts how the different fields and functions fit together. We further give some examples below showing how the pieces fit together in practice.

To access a pointer $a :: word32\ ptr$, we would use the expression

$$\text{gets}\ (\lambda s.\ \text{h\_val}\ (\text{hrs\_mem}\ (\text{t\_hrs\_}'\ s))\ a)$$

Here, the call to t_hrs_′ extracts the heap from the global state, hrs_mem fetches the raw byte component of the heap, and finally h_val decodes the bytes at pointer a into a *word32* object. Similarly, to write the value v to pointer $a :: word32\ ptr$, we would use the line

$$\text{modify}\ (\text{t\_hrs\_}'\text{\_update}\ (\text{hrs\_mem\_update}\ (\text{heap\_update}\ a\ v)))$$

Here, the sequence of updates is simply the reverse of the sequence of accessors used in the previous example.

C programs cannot freely read and write to arbitrary pointers, but instead have certain restrictions on what constitutes a valid pointer. For instance, valid accesses must not read or write to the address `NULL`, and must also be correctly aligned. To ensure that pointer accesses are valid, Norrish's C-to-Isabelle parser inserts guard statements of the form Guard $(\lambda s.\ \text{c\_guard}\ p)$ prior to each pointer p being dereferenced to ensure that the pointer is valid. c_guard is defined as follows:

$$\begin{aligned}
\text{c_guard}\ (p :: 'a\ ptr) &\equiv \text{ptr_aligned}\ p \wedge \text{c_null_guard}\ p \\
\text{ptr_aligned}\ (p :: 'a\ ptr) &\equiv \text{align_of}\ \text{TYPE}('a)\ \text{dvd unat}\ (\text{ptr_val}\ p) \\
\text{c_null_guard}\ (p :: 'a\ ptr) &\equiv 0 \notin \{\text{ptr_val}\ p\ ..+\ \text{size_of}\ \text{TYPE}('a)\}
\end{aligned}$$

In these definitions, the pointer p has type $'a\ ptr$. Each object of type $'a$ has a corresponding size in bytes size_of $\text{TYPE}('a)$ and each type has an alignment align_of $\text{TYPE}('a)$. For example, for the *word32* type, size_of $\text{TYPE}(word32) = 4$ and align_of $\text{TYPE}(word32) = 4$. In the above definitions, the function ptr_val of type $'a\ ptr \Rightarrow word32$ converts pointers into their raw *word32* values, while unat converts word types into natural numbers.

The predicate ptr_aligned ensures that a pointer p referencing an object of type $'a$ is correctly aligned for that type. Similarly, the predicate c_null_guard ensures that a pointer p is not `NULL` and does not reference an object that would overlap the `NULL` address by wrapping around the end of memory.

Figure 7.3 shows AutoCorres' attempt to abstract the `swap` program shown in Figure 7.1. Although AutoCorres has abstracted the Simpl to some extent—such as converting to a monadic form and simplifying the program's control flow structure—the model of the heap remains the same as the original Simpl. AutoCorres has blindly translated all Simpl statements modifying global memory straight into its output without any further modifications.

```
do guard (λs. c_guard a);
   t ← gets (λs. h_val (hrs_mem (t_hrs_' s)) a);
   guard (λs. c_guard b);
   modify
      (λs. t_hrs_'_update
             (hrs_mem_update
                (heap_update a (h_val (hrs_mem (t_hrs_' s)) b)))
             s);
   modify (t_hrs_'_update (hrs_mem_update (heap_update b t)))
od
```

Figure 7.3: The `swap` function as translated by AutoCorres without any changes to the heap model taking place.

7.1.2 Working with a byte-level heap

While the byte-level heap model is conceptually simple, it is unfortunately rather difficult to work with. For example, we may wish to prove a Hoare triple stating that if two pointers are passed into the function, their values will be swapped.

Our first attempt is to write the Hoare triple as follows:

$$\begin{array}{l}\{\lambda s.\ \mathsf{h\_val}\ (\mathsf{hrs\_mem}\ (\mathsf{t\_hrs\_}'\ s))\ a = v_a\ \wedge \\ \qquad \mathsf{h\_val}\ (\mathsf{hrs\_mem}\ (\mathsf{t\_hrs\_}'\ s))\ b = v_b\} \\ \quad \mathsf{swap}'\ a\ b \\ \{\lambda rv\ s.\ \mathsf{h\_val}\ (\mathsf{hrs\_mem}\ (\mathsf{t\_hrs\_}'\ s))\ a = v_b\ \wedge \\ \qquad \mathsf{h\_val}\ (\mathsf{hrs\_mem}\ (\mathsf{t\_hrs\_}'\ s))\ b = v_a\}\end{array}$$

To simplify Tuch's notation a little, we introduce two definitions, read_bytes and write_bytes, which represent reading and writing objects into the heap at a given location, defined as follows:

$$\mathsf{read\_bytes}\ p\ s \equiv \mathsf{h\_val}\ (\mathsf{hrs\_mem}\ s)\ p$$
$$\mathsf{write\_bytes}\ p\ v\ s \equiv \mathsf{hrs\_mem\_update}\ (\mathsf{heap\_update}\ p\ v)\ s$$

This makes our goal a little clearer:

$$\begin{array}{l}\{\lambda s.\ \mathsf{read\_bytes}\ a\ (\mathsf{t\_hrs\_}'\ s) = v_a\ \wedge \\ \qquad \mathsf{read\_bytes}\ b\ (\mathsf{t\_hrs\_}'\ s) = v_b\} \\ \quad \mathsf{swap}'\ a\ b \\ \{\lambda rv\ s.\ \mathsf{read\_bytes}\ a\ (\mathsf{t\_hrs\_}'\ s) = v_b\ \wedge \\ \qquad \mathsf{read\_bytes}\ b\ (\mathsf{t\_hrs\_}'\ s) = v_a\}\end{array}$$

This Hoare triple reads as follows: If the *word32* at pointer $a :: \mathit{word32\ ptr}$ contains the value v_a, and the *word32* at b contains the value v_b; then after running $\mathsf{swap}'\ a\ b$, the *word32* values at a and b will now be swapped.

This statement is not correct as written, however. For the postcondition to hold, the precondition must be strengthened to ensure that: (*i*) the pointers a and b are aligned to a 4-byte boundary; (*ii*) the pointers a and b are not `NULL`; (*iii*) the pointers a and b do not reference objects that wrap around the end of the address space; and (*iv*) the

pointers a and b do not reference objects that *partially* overlap (though if the pointers are equal, the function remains correct). The first three of these additional conditions are required by the C standard, while the fourth is required for the postcondition to hold.[2]

Taking these additional preconditions into account, the correct Hoare triple for this function is

$$\begin{array}{l}
\{\lambda s.\ \mathsf{read_bytes}\ a\ (\mathsf{t_hrs_}'\ s) = v_a\ \wedge \\
\qquad \mathsf{read_bytes}\ b\ (\mathsf{t_hrs_}'\ s) = v_b\ \wedge \\
\qquad \mathsf{c_guard}\ a\ \wedge\ \mathsf{c_guard}\ b\ \wedge \\
\qquad (a \neq b \longrightarrow \{\mathsf{ptr_val}\ a\ ..+\ \mathsf{size_of}\ \mathsf{TYPE}(\mathit{word32})\} \cap \\
\qquad\qquad \{\mathsf{ptr_val}\ b\ ..+\ \mathsf{size_of}\ \mathsf{TYPE}(\mathit{word32})\} = \emptyset)\} \\
\quad \mathsf{swap}'\ a\ b \\
\{\lambda rv\ s.\ \mathsf{read_bytes}\ a\ (\mathsf{t_hrs_}'\ s) = v_b\ \wedge \\
\qquad \mathsf{read_bytes}\ b\ (\mathsf{t_hrs_}'\ s) = v_a\}
\end{array}$$

Here, the two c_guard preconditions ensure that the pointers are correctly aligned and non-`NULL`, while the final precondition ensures that the two regions of memory don't partially overlap. With this strengthened precondition, this Hoare triple can now be proven correct, with a little manual reasoning showing that updating parts of the heap disjoint to a read don't affect that read.

If even this simple function requires such an involved specification, we are going to struggle verifying anything with significantly more complexity. A better approach is needed.

7.2 Lifting the heap

In the previous section, we saw that reasoning using a simple byte-level memory model—while possible—is rather a tedious process. We would ideally like to allow the end-user of AutoCorres to reason using a high-level memory model, abstracting over details such as unaligned pointers or partially overlapping objects. As with previous AutoCorres phases, we would also like the abstract model we generate to be provably sound; that is, a correctness result on the abstract heap model should imply that the byte-level program is also correct.

Tuch et al.'s *heap lifting* framework [103] provides some key ideas that we can use to carry out a sound abstraction. In particular, Tuch et al.'s framework provides powerful reasoning tools that can be manually applied by users to perform high-level reasoning about a byte-level model of the heap. Because AutoCorres builds upon the same memory model used by Tuch et al., users are already able to use Tuch et al.'s manual reasoning framework on AutoCorres' output. Our goal, however, is not to

[2] Low-level language aficionados will observe that in this simple example not all the preconditions are strictly required: for instance, when swapping *word32* values, if the pointers a and b are both aligned, then they can't wrap around the edge of memory, nor can they partially overlap. However, in more complex examples—such as swapping larger `structs`—all of these preconditions are required.

```
unsigned *allocate_word32(void)
{
    unsigned *result = malloc(sizeof(unsigned));
    if (result != NULL) {
        /* Tag the bytes at "*result" as a "word32". */
        /** AUXUPD: "(True, ptr_retyp \<acute>result)" */
    }
    return result;
}
```

```
allocate_word32' ≡
    do ret' ← malloc' (of_nat (size_of TYPE(word32)));
       retval ← return (ptr_coerce ret');
       when (retval ≠ NULL)
           (modify (t_hrs_'_update (hrs_htd_update (ptr_retyp retval))));
       return retval
    od
```

Figure 7.4: An example source code type annotation. If the return value of `malloc` is non-`NULL`, it is tagged using a `AUXUPD` annotation. In particular, the memory at pointer value `result` is tagged to be of type `unsigned *`; this choice of tag is determined by the type of the pointer `result`. The notation "`\<acute>`" in the `AUXUPD` annotation is the raw Isabelle notation for ‘result; that is, the value of the `result` local variable.

provide users with powerful reasoning tools for dealing with a byte-level heap—instead, we want to avoid exposing them to the low-level byte model *at all*.

To achieve this goal, we implement a simplification of Tuch et al.'s logic, more suitable for mechanised reasoning. We then use this simplified framework to automatically abstract the C-to-Isabelle parser's byte-level heap into an more abstract representation. The end result is that the user can carry out sound reasoning on a split-heap, without having to understand any of the machinery that was used to produce it.

In the rest of this section, we describe our implementation of a simplified version of Tuch et al.'s framework, and then move on to describe how this framework is used by AutoCorres.

7.2.1 Annotating the heap

In Tuch et al.'s heap lifting model, the user is required to tag bytes in the heap, specifying what type each byte should be interpreted as.[3] These tags take the form of ghost state annotations, and are added to the C source code in the form of specially parsed comments. Each address in memory can be marked as either the first byte of a C type, such as an `int` or `struct node`; the *footprint* of an earlier type, where the

[3] These type tags act as a commitment; that is, the user commits that they will only access these bytes as the type they have specified. Users may change type tags (and hence, their commitment) freely, but at any point in time each byte will only have one possible interpretation.

address simply continues a previous type; or untyped memory. We introduce a datatype *heap_typ_contents* representing each of these possibilities:

datatype *heap_typ_contents* =
HeapType *typ_uinfo*
| HeapFootprint
| HeapEmpty

The HeapType constructor of this datatype takes a parameter of type *typ_uinfo*, which is a deeply embedded description of an Isabelle/HOL type. Such Isabelle/HOL types are converted into *typ_uinfo* types using the function

typ_uinfo_t :: *'a itself* ⇒ *typ_uinfo*

The function typ_uinfo_t has overloaded definitions—that is, one definition for each Isabelle type that has a corresponding C type—which give information about the C encoding of the Isabelle type. The *typ_uinfo* returned by typ_uinfo_t is used as the tag for the HeapType constructor. For example, the first byte of a *word32* object on the heap would have the tag

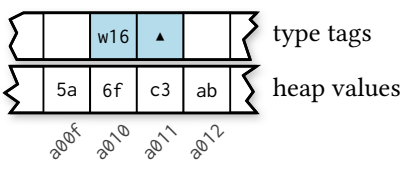
HeapType (typ_uinfo_t TYPE(*word32*))

These type annotations are stored in the hrs_htd component of the t_hrs_′ field of the global state, previously depicted in Figure 7.2. The type of the hrs_htd component is somewhat involved:

hrs_htd (t_hrs_′ *s*) :: *word32* ⇒ *bool* × (*nat* ⇒ (*typ_uinfo* × *bool*) *option*)

The hrs_htd component is a function that, for each byte in the heap, defines a *heap type descriptor* for that byte. Each such descriptor consists of a tuple of two components: the first half is a *bool* that is true if the input memory address is mapped. The second half of the tuple is a *type slice*. The type slice states what types the current byte can be validy interpreted as,[4] and whether the address is the first byte of that type or merely the footprint of a previous byte.

For example, in the following heap diagram, the two bytes starting from address `0xa010` are tagged as `signed shorts` (abbreviated `w16`), while the other bytes are untagged:



The corresponding hrs_htd function for these four addresses would be as follows:

[4] Individual bytes in Tuch et al.'s framework may have multiple different interpretations due to the nesting of C `structs`.

$$\text{hrs\_htd}\ (\text{t\_hrs\_}'\ s)\ p\ =$$
$$\begin{cases} (\text{False},\ \lambda n.\ \text{None}) & p = \texttt{0xa00f} \\ (\text{True},\ (\lambda n.\ \text{None})(0 := \text{Some}\ (\text{typ\_uinfo\_t}\ \text{TYPE}(\mathit{word16}),\ \text{True}))) & p = \texttt{0xa010} \\ (\text{True},\ (\lambda n.\ \text{None})(0 := \text{Some}\ (\text{typ\_uinfo\_t}\ \text{TYPE}(\mathit{word16}),\ \text{False}))) & p = \texttt{0xa011} \\ (\text{False},\ \lambda n.\ \text{None}) & p = \texttt{0xa012} \end{cases}$$

Much of the complexity within the hrs_htd function stems from Tuch et al.'s original reasoning framework, which used these constructs to implement a separation logic directly on the bytes of the heap. We do not need this complexity for our purposes, so simply define a function

$$\begin{aligned} \text{heap\_type\_tag} :: (\mathit{word32} \Rightarrow \mathit{bool} \times (\mathit{nat} \Rightarrow (\mathit{typ\_uinfo} \times \mathit{bool})\ \mathit{option})) \\ \Rightarrow \mathit{word32} \Rightarrow \mathit{heap\_typ\_contents} \end{aligned}$$

that translates Tuch et al.'s type tag to our simpler *heap_typ_contents* datatype.[5] When we combine the function heap_type_tag with the hrs_htd function, we end up with a function that maps from heap addresses to *heap_typ_contents*:

$$\text{heap\_type\_tag}\ (\text{hrs\_htd}\ (\text{t\_hrs\_}'\ s)) :: \mathit{word32} \Rightarrow \mathit{heap\_typ\_contents}$$

For instance, the heap in the previous example has the following definition using *heap_typ_contents*:

$$\text{heap\_type\_tag}\ (\text{hrs\_htd}\ (\text{t\_hrs\_}'\ s))\ p\ =$$
$$\begin{cases} \text{HeapEmpty} & p = \texttt{0xa00f} \\ \text{HeapType}\ (\text{typ\_uinfo\_t}\ \text{TYPE}(\mathit{word16})) & p = \texttt{0xa010} \\ \text{HeapFootprint} & p = \texttt{0xa011} \\ \text{HeapEmpty} & p = \texttt{0xa012} \end{cases}$$

Annotating the source code

A few questions remain, however. How do we add type annotations to our program? How can we specify the type of memory allocated by `malloc`? What do we do if we want to 'recycle' bytes as another type?

The type annotations are controlled directly in the C code using specially-formatted comments, which are then parsed by the C-to-Isabelle parser. These comments take the form:

```
/** AUXUPD: "(g, f)" */
```

[5] If multiple types exist for a particular byte in the original hrs_htd component of the heap, heap_type_tag simply selects the outer-most type.

Here, g is an expression that will be translated into a Simpl Guard statement,[6] while f is an expression that takes the existing hrs_htd component of the state and updates the type tag of one or more bytes to a new value.

Such type annotations are required only after addresses in memory change the type they should be interpreted as. In a standard C program, this will typically only occur after calls to functions such as `malloc` and `free`, when memory is reinterpreted through a pointer cast, or when accessing different members of a `union`. Figure 7.4 shows an example of such an annotation, both in the source code and the output of AutoCorres with heap abstraction disabled.

7.2.2 Lifting the heap

Tuch et al. defines a class of functions heap_lift that projects the byte-level heap of type *word32* ⇒ *word8* into a partial object-level heap, having type *'a ptr* ⇒ *'a option*. Our simplified definition of Tuch's heap lifting function is as follows:

$$\begin{aligned}&\text{heap\_lift } s\ (p :: 'a\ ptr) \equiv \\ &\qquad \text{if type\_tag\_valid } s\ p \wedge \text{c\_guard } p \\ &\qquad \text{then Some } (\text{read\_bytes } p\ s) \text{ else None}\end{aligned}$$

The predicate type_tag_valid in this definition determines if the heap in state s at location p is correctly tagged for the type of pointer p. It is defined as follows:

$$\begin{aligned}&\text{type\_tag\_valid } s\ (p :: 'a\ ptr) \equiv \\ &\qquad ((\text{heap\_type\_tag } (\text{hrs\_htd } s)\ (\text{ptr\_val } p)) \\ &\qquad\quad = \text{HeapType } (\text{typ\_uinfo\_t TYPE}('a))) \wedge \\ &\qquad (\forall y.\ (y \in \{\text{ptr\_val } p +_w 1\ ..+\ \text{size\_of TYPE}('a) - 1\}) \\ &\qquad\quad \longrightarrow \text{heap\_type\_tag } (\text{hrs\_htd } s)\ y = \text{HeapFootprint})\end{aligned}$$

In the lifted heap, a particular address contains a valid object if and only if (*i*) the entire range of addresses occupied by the object are correctly tagged; (*ii*) the pointer being accessed is correctly aligned; and (*iii*) the pointer is not `NULL` and does not wrap around the end of the address space. If any of these conditions fail to hold, the address resolves to None. Figure 7.5 depicts this projection.

We can derive simplified versions of reasoning Tuch et al.'s rules using our own simplified definitions. For instance, our definition of the projected heap immediately gives rise to Tuch et al.'s rule:

$$\frac{\text{heap\_lift } s\ p = \text{Some } v}{\text{c\_guard } p}$$

That is, if a heap location value is non-None on the lifted heap, then the pointer is valid on the concrete heap.

[6] We don't use the guard parameter g in our work, but it could be used, for instance, to assert that the previous type was a particular value.

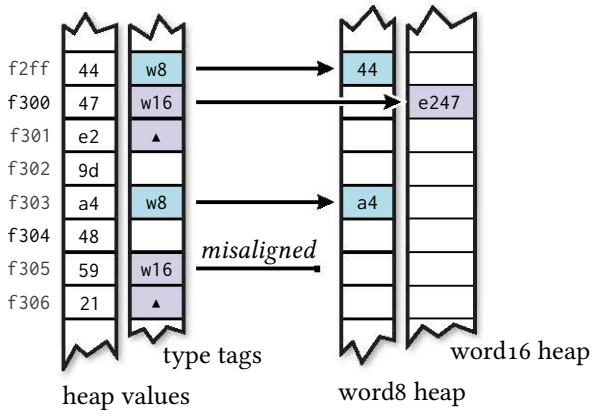


Figure 7.5: The heap lifting function.

Moreover, the user can reason that objects in the projected heap cannot partially overlap other objects of the same type:

$$\frac{\begin{array}{c}\mathsf{heap\_lift}\ s\ (a :: {'a}\ ptr) = \mathsf{Some}\ v_a \\ \mathsf{heap\_lift}\ s\ (b :: {'a}\ ptr) = \mathsf{Some}\ v_b\end{array}}{\begin{array}{l}a = b \vee \{\mathsf{ptr\_val}\ a\ ..{+}\ \mathsf{size\_of}\ \mathsf{TYPE}({'a})\} \cap \\ \qquad \{\mathsf{ptr\_val}\ b\ ..{+}\ \mathsf{size\_of}\ \mathsf{TYPE}({'a})\} = \emptyset\end{array}}$$

nor can objects of different types overlap at all:

$$\frac{\begin{array}{c}\mathsf{heap\_lift}\ s\ (a :: {'a}\ ptr) = \mathsf{Some}\ v_a \\ \mathsf{heap\_lift}\ s\ (b :: {'b}\ ptr) = \mathsf{Some}\ v_b \\ \mathsf{typ\_uinfo\_t}\ \mathsf{TYPE}({'a}) \neq \mathsf{typ\_uinfo\_t}\ \mathsf{TYPE}({'b})\end{array}}{\begin{array}{l}\{\mathsf{ptr\_val}\ a\ ..{+}\ \mathsf{size\_of}\ \mathsf{TYPE}({'a})\} \cap \\ \{\mathsf{ptr\_val}\ b\ ..{+}\ \mathsf{size\_of}\ \mathsf{TYPE}({'b})\} = \emptyset\end{array}}$$

The proof for both of these rules stem from the observation that for two objects to partially overlap, one object would have to be incorrectly tagged. Because only correctly tagged objects are present in the lifted heap, the objects cannot overlap.

With these properties in hand, we can finally derive one of the main theorems from Tuch et al.'s work, which shows that writes to valid addresses are equivalent to functional updates on the projected heap:

$$\frac{\mathsf{heap\_lift}\ s\ p = \mathsf{Some}\ v'}{\mathsf{heap\_lift}\ (\mathsf{write\_bytes}\ p\ v\ s) = (\mathsf{heap\_lift}\ s)(p := \mathsf{Some}\ v)}$$

With our definition heap_lift and this rule, we can (*i*) lift the byte-level heap—containing the bytes of every type of object—into multiple abstract heaps, one for each different type; (*ii*) read objects from the heap by accessing the appropriate heap; and finally (*iii*) write objects to the heap by carrying out functional updates.

Reasoning at the level of lifted heaps greatly simplifies proofs interacting with the

heap. The correctness statement of our `swap` function becomes:

$$\begin{array}{l}\{\lambda s.\ \mathsf{heap\_lift}\ (\mathsf{t\_hrs\_}'\ s)\ a = \mathsf{Some}\ v_a\ \wedge \\ \qquad \mathsf{heap\_lift}\ (\mathsf{t\_hrs\_}'\ s)\ b = \mathsf{Some}\ v_b\} \\ \quad \mathsf{swap}'\ a\ b \\ \{\lambda rv\ s.\ \mathsf{heap\_lift}\ (\mathsf{t\_hrs\_}'\ s)\ a = \mathsf{Some}\ v_b\ \wedge \\ \qquad \mathsf{heap\_lift}\ (\mathsf{t\_hrs\_}'\ s)\ b = \mathsf{Some}\ v_a\}\end{array}$$

This result is proved by unfolding the definition of swap′, executing a VCG, and running Isabelle/HOL's `auto` tactic with the above rules.

7.2.3 Limitations of the heap lifting approach

Tuch et al.'s original heap lifting framework relies heavily on Isabelle/HOL's simplifier to automatically apply recursive conditional rewrite rules, so that low-level C operations are rewritten into high-level heap updates.[7]

As programs become more complex, so does application of the lifting predicates. For instance, consider Suzuki's challenge [97] to prove that the following fragment returns `4` under the assumption that the four pointers `w`, `x`, `y` and `z` are distinct:

```
w->next = x; x->next = y; y->next = z; x->next = z;
w->data = 1; x->data = 2; y->data = 3; z->data = 4;
return w->next->next->data;
```

In this fragment of code, Isabelle/HOL times out while attempting to apply the heap lifting rules described above. The primary problem is the deep nesting of write operations, preventing Isabelle's simplifier from identifying which rewrite rules to apply, because their recursive preconditions become too large and too deep. Basically, at even a moderately large scale, the prover becomes overloaded simply applying heap abstraction rules, and never proceeds to reasoning about the actual semantics of the program.

Tuch's framework also suffers the problem that it is awkward to state what *doesn't* change during a function's execution. For instance, in our `swap` example we might like to say that the two input pointers are swapped in memory, and *nothing else* changes. The difficulty of writing this statement using Tuch's framework arises because of Isabelle/HOL's limited ability to quantify over types. While we can say that lifted heaps of a particular type remain unchanged, it is harder to say that lifted heaps of *all* types remain unchanged without imposing side conditions on the user that must be manually discharged.[8]

Ad hoc heap lifting is also unsatisfactory on a more fundamental usability level: while C programs need byte-level access to memory on occasion, most C functions are type-safe. Ideally, for the majority of type-safe code, we should present the user with a specification that operates directly on the lifted heap instead of requiring the user to manually appeal to heap abstraction predicates.

[7] Our simplified implementation of Tuch et al.'s framework suffers the same problem.

[8] Tuch tackled the problem of writing specifications which deal with specifying what *doesn't* change on the heap when a function executes by using a separation logic implementation on top of his framework [100]. While this solves the problem, it is rather a heavy-weight solution. Ideally there would be a simpler solution for a trivial function such as `swap`.

7.3 Automated state abstraction

The approach we propose in this chapter to simplify reasoning about pointer programs is to add an abstraction step to AutoCorres where we automatically translate the program to use a split-heap model instead of a byte-level heap model. We call this process *heap abstraction*. This involves taking the *globals* record generated by the C-to-Isabelle parser to represent the program's global state, and generating a new record *lifted_globals* that contains a separate heap for each type used in the program. We finally rewrite the input program specification to operate on this new abstract state type. Reasoning on such an abstracted state removes the need to invoke lifting rewrite rules, instead using standard Isabelle/HOL mechanisms such as functional updates to reason about the heap.

Because heap abstraction hides the byte-level view of the heap, low-level reasoning on heap abstracted functions is no longer possible. To ensure that users still have the flexibility to reason about functions that deliberately violate type-safety, such as `memset` or `memcpy`, we also introduce mechanisms that allow heap-abstracted functions to call into byte-level functions, and *vice versa*. Using these mechanisms, we can allow the user to opt-out of heap abstraction on a per-function basis.

This section introduces the refinement framework and rules used to carry out the abstraction. The following sections provide examples using the resulting translation, and demonstrate how heap abstraction can interact with type-unsafe code.

7.3.1 Generating the abstract state type

We start the process of heap abstraction by generating an appropriate abstract state type for our program. This is done by analysing the source program to determine which types the program accesses on the heap, i.e., which pointer types are used as arguments to the h_val and heap_update functions described earlier. For each heap type $'a$ required, we place two functions into the *lifted_globals* record: an is_valid_x function of type $'a\ ptr \Rightarrow bool$ and a heap_x function of type $'a\ ptr \Rightarrow 'a$, where x is the name of the type $'a$. The former function determines if a particular address contains a valid value (that is, $\exists x.$ heap_lift $s\ p =$ Some x), while the latter function contains the logical value of each address (the (heap_lift $s\ p$)). For example, in a program that uses `unsigned long` pointers and `struct node` pointers, the generated *lifted_globals* record would contain the following fields:

record *lifted_globals* =
  is_valid_w32 :: *word32 ptr* ⇒ *bool*
  heap_w32 :: *word32 ptr* ⇒ *word32*
  is_valid_node_C :: *node_C ptr* ⇒ *bool*
  heap_node_C :: *node_C ptr* ⇒ *node_C*
  ...

While splitting data and validity information initially appears more complex than simply having a partial function $'a\ ptr \Rightarrow 'a\ option$, we have found that this approach allows a clearer separation between *what* data is contained at an address, and *which*

addresses are valid. Further, while the data at a particular address frequently changes, the validity of an address rarely changes. Splitting the two dimensions makes it clear that changes to one are independent of the other.

7.3.2 Ingredients for generating the abstract program

Once we have a new abstract state for our program to use, we next translate the actual program to use this generated state. Informally, when abstracting a program from the byte-level heap to an object-level heap, three primary cases must be handled:

Heap reads: Operations on the concrete specification that access the heap, such as h_val, need to be abstracted into an access of the equivalent abstract function. One difficulty is that for many concrete operations, the abstract equivalent is only sound given certain side conditions: a h_val is only equivalent to a functional access if the pointer being read from is valid. To resolve this, when translating an expression we emit suitable guard statements prior to each translated expression.

Heap writes: Operations that modify the heap are handled similarly to heap reads. A heap_update operation becomes a functional update on the appropriate heap, again with guards emitted to deal with the side condition that the pointer being written to is valid.

Guard statements: c_guard assertions in the concrete specification are abstracted into accesses of the appropriate is_valid_x function.

We describe how each of these three cases are formally abstracted in the next section.

Formalising heap abstraction

To generate the actual abstract program, we use a syntax-directed system of rules to convert statements and expressions in our concrete program to their abstract equivalents, similar to that used in Chapter 6. That is, we simultaneously generate both the abstract version of the program and a proof that the abstraction is correct.

The top-level statement we use to both generate and prove the abstraction is a refinement predicate, similar to that used in previous chapters. The predicate for heap abstraction is called $\text{corres}_{\text{HA}}$, and is defined as follows:

$$
\begin{aligned}
&\text{corres}_{\text{HA}}\ st\ A\ C \equiv \\
&\quad \forall s.\ \neg\ \text{failed}\ (A\ (st\ s)) \longrightarrow \\
&\qquad (\forall (r,\ t) \in \text{results}\ (C\ s). \\
&\qquad\quad \text{case}\ r\ \text{of} \\
&\qquad\qquad \text{Exc}\ r \Rightarrow (\text{Exc}\ r,\ st\ t) \in \text{results}\ (A\ (st\ s)) \\
&\qquad\qquad \mid \text{Norm}\ r \Rightarrow (\text{Norm}\ r,\ st\ t) \in \text{results}\ (A\ (st\ s))) \wedge \\
&\qquad \neg\ \text{failed}\ (C\ s)
\end{aligned}
$$

Informally, the $\text{corres}_{\text{HA}}$ predicate declares that a monadic program C is a refinement of an abstract program A if the following holds (and assuming that A executing

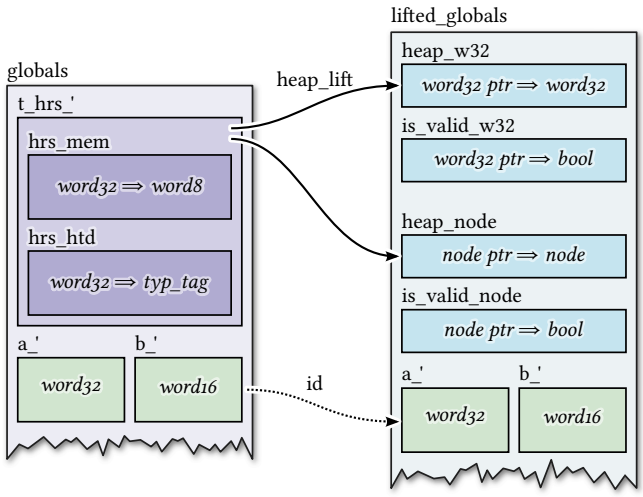


Figure 7.6: Translation from the default C-to-Isabelle parser global state to a lifted global state. Each object type used in the C sources is given functions of the form heap_*x* and is_valid_*x*. The former maps a pointer to a high-level object, while the latter maps a pointer to a boolean indicating if the address contains a valid object. Global variables (in this example, a_′ and b_′) are mapped unchanged into the new state type.

from the state *st s* does not fail): (*i*) for each normal execution of *C*, there is an equivalent execution of *A* which has a corresponding abstract state; (*ii*) for each exceptional execution of *C*, *A* similarly has a corresponding abstract execution; and (*iii*) under the assumption that *A* has not failed, neither will *C*.

The first of these conditions states that program *A* produces a superset of states of program *C*. Thus, if a property holds for all states in program *A*, then we can reason that it also holds on program *C*. The second condition allows us to reason that program *C* will never fail by proving that program *A* never fails. With these two conditions, we can typically prove useful properties about our original concrete program without needing to ever reason on it directly, as we will demonstrate in Chapter 8.

The state translation parameter *st* of the above predicate needs to convert the byte-level state *globals* into our newly generated type *lifted_globals*. Internally it does this by using the heap_lift function described earlier. For example, in a program that uses `unsigned int` pointers and `struct node` pointers, and has two global variables `a` and `b`, the generated state translation function st would be:

$$
\begin{aligned}
&\mathsf{st}\ s \equiv \\
&\qquad (\!|\, \mathsf{is\_valid\_w32} = \lambda p.\ \exists x.\ \mathsf{heap\_lift}\ (\mathsf{t\_hrs\_}'\ s)\ p = \mathsf{Some}\ x, \\
&\qquad\quad \mathsf{heap\_w32} = \lambda p.\ \mathsf{the}\ (\mathsf{heap\_lift}\ (\mathsf{t\_hrs\_}'\ s)\ p), \\
&\qquad\quad \mathsf{is\_valid\_node\_C} = \lambda p.\ \exists x.\ \mathsf{heap\_lift}\ (\mathsf{t\_hrs\_}'\ s)\ p = \mathsf{Some}\ x, \\
&\qquad\quad \mathsf{heap\_node\_C} = \lambda p.\ \mathsf{the}\ (\mathsf{heap\_lift}\ (\mathsf{t\_hrs\_}'\ s)\ p), \\
&\qquad\quad \mathsf{a\_}' = \mathsf{a\_}'\ s, \\
&\qquad\quad \mathsf{b\_}' = \mathsf{b\_}'\ s\ |\!)
\end{aligned}
$$

The is_valid_*x* and heap_*x* functions are derived from the t_hrs_′ component of the original heap, while the global variables a_′ and b_′ are copied from the *globals* record into the *lifted_globals* record unchanged. This is depicted in Figure 7.6.

During the abstraction process, we also need to abstract expressions and state-modification statements. To achieve this, we define two new predicates. The first predicate $\text{abs\_expr}_{\text{HA}}$ states that a particular abstract expression corresponds to a concrete expression, while the second predicate $\text{abs\_modifies}_{\text{HA}}$ asserts that an abstract state-modifying statement corresponds to a concrete equivalent:

$$\begin{aligned}&\text{abs\_expr}_{\text{HA}}\ st\ P\ a\ c \equiv \\ &\qquad \forall s.\ P\ (st\ s) \longrightarrow c\ s = a\ (st\ s)\end{aligned}$$

$$\begin{aligned}&\text{abs\_modifies}_{\text{HA}}\ st\ P\ a\ c \equiv \\ &\qquad \forall s.\ P\ (st\ s) \longrightarrow st\ (c\ s) = a\ (st\ s)\end{aligned}$$

Both functions take a state translation function *st* and a precondition *P* on the abstract state *st s*. The first function, $\text{abs\_expr}_{\text{HA}}$, states that the expression *a*, when given an abstract state *st s*, matches the concrete expression *c* given a concrete state *s*. The second function, $\text{abs\_modifies}_{\text{HA}}$, states that abstracting a state and then running *a* on the result is the same as running *c* on the original state and then abstracting the result.

The first function $\text{abs\_expr}_{\text{HA}}$ is used to declare that two expressions (that possibly read the global state) are *equal*; while $\text{abs\_modifies}_{\text{HA}}$ declares that two functions that return a new state are *equivalent with respect to st*. The first is useful for values that are not expected to change after abstraction, such as the result of numeric calculations, while the second is useful for statements that modify the state, such as writing to a pointer.

For example, we can write that the following expressions that model the C expression `*p + 1` are equal (under the precondition that *p* is valid):

$$\begin{aligned}&\text{abs\_expr}_{\text{HA}}\ \text{st}\ (\lambda s.\ \text{is\_valid\_w32}\ s\ p) \\ &\quad (\lambda s.\ \text{heap\_w32}\ s\ (p :: \mathit{word32\ ptr}) +_{\text{w}} 1) \\ &\quad (\lambda s.\ \text{read\_bytes}\ p\ (\text{t\_hrs\_}'\ s) +_{\text{w}} 1)\end{aligned}$$

Additionally, we can write that the following state modification statements that model the C statement `*p = 42` result in equivalent states (again, under the precondition that *p* is valid):

$$\begin{aligned}&\text{abs\_modifies}_{\text{HA}}\ \text{st}\ (\lambda s.\ \text{is\_valid\_w32}\ s\ p) \\ &\quad (\text{heap\_w32\_update}\ (\lambda h.\ h(p := \text{0x2A}))) \\ &\quad (\text{t\_hrs\_}'\text{\_update}\ (\text{write\_bytes}\ p\ \text{0x2A}))\end{aligned}$$

Writing program-agnostic rules

AutoCorres does not know in advance what heap types a given program will use, nor the exact format of the *globals* type, the *lifted_globals* type or the state translation function st. In fact, even the constants t_hrs_′ and t_hrs_′_update don't exist until the C-to-Isabelle parser has run to completion. This means that we cannot write static theorems referencing these constants. Any rule that we use for heap abstraction must

$$
\begin{aligned}
&\text{read\_write\_valid } r\ w \equiv \\
&\qquad (\forall f\ s.\ r\ (w\ f\ s) = f\ (r\ s)) \wedge \\
&\qquad (\forall s\ f.\ f\ (r\ s) = r\ s \longrightarrow w\ f\ s = s) \wedge \\
&\qquad (\forall f\ f'\ s.\ f\ (r\ s) = f'\ (r\ s) \longrightarrow w\ f\ s = w\ f'\ s) \wedge \\
&\qquad (\forall f\ g\ s.\ w\ f\ (w\ g\ s) = w\ (\lambda x.\ f\ (g\ x))\ s) \\
&\text{valid\_implies\_cguard } st\ v_r \equiv \\
&\qquad \forall s\ p.\ v_r\ (st\ s)\ p \longrightarrow \text{c\_guard } p \\
&\text{heap\_decode\_bytes } st\ v_r\ h_r\ t\_hrs_r \equiv \\
&\qquad \forall s\ p.\ v_r\ (st\ s)\ p \longrightarrow \\
&\qquad\qquad h_r\ (st\ s)\ p = \text{h\_val } (\text{hrs\_mem } (t\_hrs_r\ s))\ p \\
&\text{heap\_encode\_bytes } st\ v_r\ h_w\ t\_hrs_w \equiv \\
&\qquad \forall s\ p\ x.\ v_r\ (st\ s)\ p \longrightarrow \\
&\qquad\qquad st\ (t\_hrs_w\ (\text{hrs\_mem\_update } (\text{heap\_update } p\ x))\ s) = \\
&\qquad\qquad\quad h_w\ (\lambda f.\ f(p := x))\ (st\ s) \\
&\text{write\_preserves\_valid } v_r\ h_w \equiv \\
&\qquad \forall p\ f\ s.\ v_r\ s\ p \longrightarrow v_r\ (h_w\ f\ s)\ p
\end{aligned}
$$

Table 7.1: Formal definitions of the predicates used in the valid_heap definition.

therefore be agnostic of these details. To help achieve this, we introduce a predicate valid_heap, defined as follows:[9]

$$
\begin{aligned}
&\text{valid\_heap } st\ h_r\ h_w\ v_r\ v_w\ t\_hrs_r\ t\_hrs_w \equiv \\
&\qquad \text{read\_write\_valid } h_r\ h_w \wedge \\
&\qquad \text{read\_write\_valid } v_r\ v_w \wedge \\
&\qquad \text{read\_write\_valid } t\_hrs_r\ t\_hrs_w \wedge \\
&\qquad \text{valid\_implies\_cguard } st\ v_r \wedge \\
&\qquad \text{heap\_decode\_bytes } st\ v_r\ h_r\ t\_hrs_r \wedge \\
&\qquad \text{heap\_encode\_bytes } st\ v_r\ h_w\ t\_hrs_w \wedge \\
&\qquad \text{write\_preserves\_valid } v_r\ h_w
\end{aligned}
$$

The predicate takes seven parameters: *st* is the state translation function; t_hrs_r and t_hrs_w correspond to the yet-to-be-defined t_hrs_′ reader function and the equivalent t_hrs_′_update writer function, respectively. h_r and h_w are the heap_*x* reader and writer functions for a particular type, while v_r and v_w are the corresponding is_valid_*x* reader and writer functions for that type.

Informally, the valid_heap predicate states that (*i*) the pairs of readers and writers have 'sensible' semantics; (*ii*) if is_valid_*x* is true for a particular address, then c_guard also holds for that address; (*iii*) h_r contains correctly decoded objects for valid addresses; (*iv*) h_w writes correctly encoded objects for valid addresses; and (*v*) writing data to the heap does not affect the validity of pointers.

The formal definition of the predicates used in the definition of valid_heap are shown in Table 7.1. The predicates are as follows:

[9] Isabelle/HOL users will observe that Isabelle's *locales* feature [8] could be used here; from a proof mechanisation perspective, we found that simply having an explicit assumption was easier to implement.

- read_write_valid r w takes a field reader $r :: 'r \Rightarrow 'a$, which extracts a field of type $'a$ from a record of type $'r$, and a field writer w, which is given a function that updates the field's existing value, and returns a new record. The read_write_valid predicate states that: (*i*) reading from a just-written field returns that same value; (*ii*) writing an existing value to a field causes the record to be unchanged; and (*iii*) writing the same field value to the same original record will result in the same new record.
- valid_implies_cguard states that an abstract address is only valid if the concrete pointer satisfies the c_guard predicate.
- heap_decode_bytes and heap_encode_bytes together state that reads and writes to valid abstract addresses correspond to the appropriate h_val and heap_update operations on the concrete heap.
- write_preserves_valid states that heap updates do not affect the value read out of the is_valid_x functions.

When we write our heap abstraction rules, we assume that the valid_heap predicate is true, which in turn lets us assume key properties required of functions associated with it. After AutoCorres has generated the *lifted_globals* type (along with its associated is_valid_x and heap_x functions and state translation function st), it will then proceed to show that the valid_heap predicate holds for each type. Once proven, the entire set of heap abstraction rules can be used.

For example, the following rule states that a c_guard statement on the concrete heap can be abstracted into a check of the is_valid_x predicate (i.e., v_r) on the abstract heap:

$$\frac{\text{valid\_heap}\ st\ h_r\ h_w\ v_r\ v_w\ t\_hrs_r\ t\_hrs_w}{\text{corres}_{\text{HA}}\ st\ (\text{guard}_{\text{E}}\ (\lambda s.\ v_r\ s\ p))\ (\text{guard}_{\text{E}}\ (\lambda s.\ \text{c\_guard}\ p))}$$

When AutoCorres' heap abstraction phase begins, it will instantiate this rule once for each generated heap, using the automatically proven valid_heap predicates. From our example heap containing the types `unsigned long` and `struct node`, this will result in the two rules:

$$\begin{aligned}&\text{corres}_{\text{HA}}\ \text{st}\\ &\quad(\text{guard}_{\text{E}}\ (\lambda s.\ \text{is\_valid\_w32}\ s\ p))\\ &\quad(\text{guard}_{\text{E}}\ (\lambda s.\ \text{c\_guard}\ (p :: \mathit{word32}\ \mathit{ptr})))\end{aligned}$$

$$\begin{aligned}&\text{corres}_{\text{HA}}\ \text{st}\\ &\quad(\text{guard}_{\text{E}}\ (\lambda s.\ \text{is\_valid\_node\_C}\ s\ p))\\ &\quad(\text{guard}_{\text{E}}\ (\lambda s.\ \text{c\_guard}\ (p :: \mathit{node\_C}\ \mathit{ptr})))\end{aligned}$$

The first abstracts c_guard assertions of *word32* pointers into is_valid_w32 checks, while the second does the same for the *node_C* type.

7.3.3 Heap abstraction ruleset

As described previously, AutoCorres uses a syntax-directed set of rules to carry out the conversion from a byte-level heap to an abstract object-level heap. The rules used

Statement abstraction

$$\frac{\mathsf{abs\_modifies}_{\mathsf{HA}}\ st\ P\ m\ m'}{\mathsf{corres}_{\mathsf{HA}}\ st\ (\mathbf{do}\ \mathsf{guard}_{\mathsf{E}}\ P;\ \mathsf{modify}_{\mathsf{E}}\ m\ \mathbf{od})\ (\mathsf{modify}_{\mathsf{E}}\ m')}$$
HeapAbsModify

$$\frac{\mathsf{abs\_expr}_{\mathsf{HA}}\ st\ P\ e\ e'}{\mathsf{corres}_{\mathsf{HA}}\ st\ (\mathbf{do}\ \mathsf{guard}_{\mathsf{E}}\ P;\ \mathsf{gets}_{\mathsf{E}}\ e\ \mathbf{od})\ (\mathsf{gets}_{\mathsf{E}}\ e')}$$
HeapAbsGets

$$\frac{\mathsf{abs\_expr}_{\mathsf{HA}}\ st\ P\ e\ e'}{\mathsf{corres}_{\mathsf{HA}}\ st\ (\mathsf{guard}_{\mathsf{E}}\ (\lambda s.\ P\ s \wedge e\ s))\ (\mathsf{guard}_{\mathsf{E}}\ e')}$$
HeapAbsGuard

$$\frac{\begin{array}{c}\forall r.\ \mathsf{corres}_{\mathsf{HA}}\ st\ (B\ r)\ (B'\ r) \\ \forall r.\ \mathsf{abs\_expr}_{\mathsf{HA}}\ st\ (P\ r)\ (c\ r)\ (c'\ r)\end{array}}{\begin{array}{l}\mathsf{corres}_{\mathsf{HA}}\ st \\ \quad (\mathbf{do}\ \mathsf{guard}_{\mathsf{E}}\ (P\ i); \\ \qquad \mathsf{whileLoop}_{\mathsf{E}}\ c\ (\lambda r.\ \mathbf{do}\ v \leftarrow B\ r;\ \mathsf{guard}_{\mathsf{E}}\ (P\ v);\ \mathsf{return}_{\mathsf{E}}\ v\ \mathbf{od})\ i \\ \quad \mathbf{od}) \\ \quad (\mathsf{whileLoop}_{\mathsf{E}}\ c'\ B'\ i)\end{array}}$$
HeapAbsWhile

$$\frac{\mathsf{corres}_{\mathsf{HA}}\ st\ L\ L' \quad \mathsf{corres}_{\mathsf{HA}}\ st\ R\ R' \quad \mathsf{abs\_expr}_{\mathsf{HA}}\ st\ P\ c\ c'}{\mathsf{corres}_{\mathsf{HA}}\ st\ (\mathbf{do}\ \mathsf{guard}_{\mathsf{E}}\ P;\ \mathsf{condition}_{\mathsf{E}}\ c\ L\ R\ \mathbf{od})\ (\mathsf{condition}_{\mathsf{E}}\ c'\ L'\ R')}$$
HeapAbsCond

$$\frac{\mathsf{corres}_{\mathsf{HA}}\ st\ L\ L' \quad \forall r.\ \mathsf{corres}_{\mathsf{HA}}\ st\ (R\ r)\ (R'\ r)}{\mathsf{corres}_{\mathsf{HA}}\ st\ (L \gg=_{\mathsf{E}} R)\ (L' \gg=_{\mathsf{E}} R')}$$
HeapAbsBind

$$\frac{\mathsf{corres}_{\mathsf{HA}}\ st\ L\ L' \quad \forall r.\ \mathsf{corres}_{\mathsf{HA}}\ st\ (R\ r)\ (R'\ r)}{\mathsf{corres}_{\mathsf{HA}}\ st\ (\mathsf{catch}_{\mathsf{E}}\ L\ R)\ (\mathsf{catch}_{\mathsf{E}}\ L'\ R')}$$
HeapAbsCatch

$$\mathsf{corres}_{\mathsf{HA}}\ st\ (\mathsf{throw}_{\mathsf{E}}\ a)\ (\mathsf{throw}_{\mathsf{E}}\ a)$$
HeapAbsThrow

Table 7.2: Syntax-directed rules used for heap abstraction of program statements.

Expression and state modification abstraction

$$\frac{}{\text{abs\_expr}_{\text{HA}}\ st\ (\lambda\_.\ \text{True})\ (\lambda s.\ c)\ (\lambda s.\ c)}\ \textsc{HeapAbsConstant}$$

$$\frac{\text{valid\_heap}\ st\ h_r\ h_w\ v_r\ v_w\ t\_hrs_r\ t\_hrs_w \qquad \text{abs\_expr}_{\text{HA}}\ st\ P\ e\ e'}{\begin{array}{l}\text{abs\_expr}_{\text{HA}}\ st\ (\lambda s.\ P\ s \wedge v_r\ s\ (e\ s))\\ \quad (\lambda s.\ h_r\ s\ (e\ s))\ (\lambda s.\ \text{h\_val}\ (\text{hrs\_mem}\ (t\_hrs_r\ s))\ (e'\ s))\end{array}}\ \textsc{HeapAbsExpr}$$

$$\frac{\text{valid\_heap}\ st\ h_r\ h_w\ v_r\ v_w\ t\_hrs_r\ t\_hrs_w \qquad \text{abs\_expr}_{\text{HA}}\ st\ P_b\ b\ b' \qquad \text{abs\_expr}_{\text{HA}}\ st\ P_c\ c\ c'}{\begin{array}{l}\text{abs\_modifies}_{\text{HA}}\ st\ (\lambda s.\ P_b\ s \wedge P_c\ s \wedge v_r\ s\ (b\ s))\ (\lambda s.\ h_w\ (\lambda x.\ x(b\ s := c\ s))\ s)\\ \quad (\lambda s.\ t\_hrs_w\ (\text{hrs\_mem\_update}\ (\text{heap\_update}\ (b'\ s)\ (c'\ s)))\ s)\end{array}}\ \textsc{HeapAbsModifies}$$

$$\frac{\text{valid\_heap}\ st\ h_r\ h_w\ v_r\ v_w\ t\_hrs_r\ t\_hrs_w \qquad \text{abs\_expr}_{\text{HA}}\ st\ P\ f\ f'}{\begin{array}{l}\text{abs\_expr}_{\text{HA}}\ st\ (\lambda s.\ P\ s \wedge v_r\ s\ (f\ s))\\ \quad (\lambda s.\ \text{True})\ (\lambda s.\ \text{c\_guard}\ (f'\ s))\end{array}}\ \textsc{HeapAbsGuard}$$

$$\frac{\text{abs\_expr}_{\text{HA}}\ st\ P\ b\ b' \qquad \text{abs\_expr}_{\text{HA}}\ st\ Q\ a\ a'}{\text{abs\_expr}_{\text{HA}}\ st\ (\lambda s.\ P\ s \wedge Q\ s)\ (\lambda s.\ a\ s\ (b\ s))\ (\lambda s.\ a'\ s\ \$\ b'\ s)}\ \textsc{HeapAbsFunApp}$$

Table 7.3: Syntax-directed rules used for heap abstraction of state updates and expressions.

to abstract byte-level heaps into their abstract equivalents are shown in Table 7.2 and Table 7.3.

The first table lists rules required to abstract statements. These rules do not significantly change the program structure, but simply convert the expressions inside gets_E statements, the conditions of condition and $\text{whileLoop}_\text{E}$ statements and so on, to their abstract equivalents using $\text{abs\_expr}_\text{HA}$ and $\text{abs\_modifies}_\text{HA}$.

The predicates $\text{abs\_expr}_\text{HA}$ and $\text{abs\_modifies}_\text{HA}$ both contain a precondition. For most statements, this precondition is checked by emitting a guard_E statement prior to the statement being abstracted (such as in HeapAbsModify or HeapAbsGets). In the case of the condition of $\text{whileLoop}_\text{E}$, the precondition must be checked *every* time the loop condition is evaluated; hence, a guard_E statement is emitted both before the loop is first executed, and also at the end of the loop's inner body, as shown in HeapAbsWhile.

Usually, the guard statements emitted during heap abstraction will be redundant, either because the expression being abstracted does not access the heap (and hence the precondition is $(\lambda s.\ \text{True})$), or because it has already been checked by a previous guard statement originally emitted by the C-to-Isabelle parser. In both cases, the flow-sensitive optimisations described in Section 5.2 will remove such redundant guards after heap abstraction is complete.

Table 7.3 shows rules for abstracting expressions. The rule HeapAbsConstant allows expressions that do not access the global state to be copied into the abstract output unmodified. The rules HeapAbsExpr, HeapAbsModify and HeapAbsGuard convert expressions, state-modification statements and c_guard expressions to their abstract equivalents.

The rule HeapAbsFunApp breaks compound expressions into component parts. For example, if an expression of the form $a\ b$ does not match any other rule, the rule HeapAbsFunApp will separately attempt to abstract a and b. These sub-expressions may contain concrete accesses to the byte-level heap, which can then be abstracted. Isabelle/HOL's resolution engine performs higher-order unification, which means a pattern such as *?a ?b* can unify an infinite number of ways with any other term. We use Lammich's trick [63] to tame Isabelle's resolution engine by making function application explicit by converting the term $a\ b$ to $a\ \$\ b$ (where the operator \$ is a constant indicating function application), before attempting to apply the rule HeapAbsFunApp.

7.3.4 Example: swap

Figure 7.7 shows our example swap program after heap abstraction has been applied. Here we introduced the notation $s[p]$ for accessing the heap in state s at pointer p; and $s[p := v]$ for writing the value v in state s at pointer p. The particular heap being read/modified is determined by the type of the pointer p.

As expected, the h_val and heap_update functions on the concrete heap are converted to functional accesses and updates, while the c_guard checks of pointers have been abstracted into checks on the is_valid_w32 function.

```
do guard (λs. is_valid_w32 s a);
   t ← gets (λs. s[a]);
   guard (λs. is_valid_w32 s b);
   modify (λs. s[a := s[b]]);
   modify (λs. s[b := t])
od
```

Figure 7.7: The `swap` function translated by AutoCorres with heap abstraction enabled.

Our correctness statement for swap can now be stated as follows:

$$\begin{array}{l}\{\lambda s.\ \text{is\_valid\_w32}\ s\ a \wedge s[a] = v_a \wedge \\ \qquad \text{is\_valid\_w32}\ s\ b \wedge s[b] = v_b\} \\ \quad \text{swap}'\ a\ b \\ \{\lambda rv\ s.\ s[a] = v_b \wedge s[b] = v_a\}\end{array}$$

This goal is automatically discharged by applying a VCG and running Isabelle/HOL's built-in `auto` tactic, without needing to appeal to further rules. Having eliminated the need for complex conditional rewrites, Isabelle/HOL is now able to work directly on simpler data types with all the power of its built-in automation.

We can also write a stronger version of the rule that is more precise about the effects of the function:

$$\begin{array}{l}\forall s_o.\ \{\lambda s.\ \text{is\_valid\_w32}\ s\ a \wedge \text{is\_valid\_w32}\ s\ b \wedge s = s_o\} \\ \quad \text{swap}'\ a\ b \\ \quad \{\lambda rv\ s.\ rv = () \wedge s = s_o[a := s_o[b]][b := s_o[a]]\}\end{array}$$

This rule states not only that the values of a and b are swapped, but that these are the *only* changes to the state that are made. This rule, like the previous, is solved simply by applying a VCG and the built-in `auto` tactic.

7.3.5 Example: Unsuccessfully abstracting a type-unsafe function

Heap abstraction only produces a valid abstraction if the function being abstracted is type-safe. In particular, the source program must use source annotations to commit to each byte only being accessed as a single type, as described earlier in Section 7.2.1. Section 7.5 looks at how we can reason about programs that mix calls among functions using abstract object-level heaps and functions using their original byte-level heaps, but for now it is instructive to observe what happens if we attempt to abstract a type-unsafe function.

The C function shown in Figure 7.8 takes a pointer to an unsigned integer `u`, and then uses a type-unsafe cast to update the first byte of the value at `*u` to the value `1`. Finally, it returns the new value of `*u`. On a little-endian machine,[10] running the program with an input of `1000` (`0x3e8`) will cause the program to return `769` (`0x301`).

[10] A little-endian machine stores in memory the least-significant byte of an integer value *first*; so the integer value `0x11223344` will be stored in memory as "`44 33 22 11`".

```
unsigned int type_unsafe(unsigned int *u)
{
    unsigned char *c = (unsigned char *)u;
    *c = 1;
    return *u;
}
```

```
type_unsafe' u ≡
        do c ← return (ptr_coerce u);
           guard (λs. is_valid_w8 s c);
           modify (λs. s[c := 1]);
           guard (λs. is_valid_w32 s u);
           gets (λs. s[u])
        od
```

Figure 7.8: A type-unsafe program and its (apparently) incorrect abstraction generated by AutoCorres' heap abstraction.

Abstracting the program using AutoCorres with heap abstraction enabled results in the output shown at the bottom of Figure 7.8. AutoCorres has separated the read of the `unsigned int` pointer and the write to the `unsigned char` pointer into two separate heaps. The abstracted function hence (incorrectly) returns the initial value of `*u`.

We can go on and confirm our intuition that the abstract output of AutoCorres is incorrect by proving the following theorem:

$$\begin{array}{l}\{\,\lambda s.\ \text{is\_valid\_w32}\ s\ p\ \wedge \\ \quad\ \text{is\_valid\_w8}\ s\ (\text{ptr\_coerce}\ p :: \mathit{word8\ ptr})\ \wedge\ s[p] = v\,\} \\ \ \ \text{type\_unsafe}'\ p \\ \{\,\lambda rv\ s.\ rv = v\,\}\end{array}$$

The Hoare triple states that, assuming that p is a valid *word32* pointer and also a valid *word8* pointer, then the function will return the initial (unchanged) value of heap_w32 p. This clearly does not match our observation of running the C code, so something has clearly gone wrong. How did we arrive at this unsound reasoning?

The first observation we can make is that the Hoare triple above really *is* correct if we consider the function type_unsafe′ in isolation from its origin C code; that is, the abstract function type_unsafe′ really *does* have the behaviour described in the Hoare triple above. The problem isn't in our reasoning about type_unsafe′, but that this function doesn't match our concrete C program.

The second observation we make is that no concrete state s, when abstracted through the generated state translation function st, satisfies the precondition of our Hoare triple above. In particular, there does not exist an abstract state *st s* such that is_valid_w32 and is_valid_w8 hold on the same address. We can prove this with the theorem:

$$\begin{array}{l}\nexists s.\ \text{is\_valid\_w32}\ (\text{st}\ s)\ (p :: \mathit{word32\ ptr})\ \wedge \\ \quad\ \text{is\_valid\_w8}\ (\text{st}\ s)\ (\text{ptr\_coerce}\ p :: \mathit{word8\ ptr})\end{array}$$

This is because each byte has a single type tag associated with it. On the abstract level, an object can only be valid if its associated bytes are correctly tagged. For a single address to be both a valid *word8* and a valid *word32*, it would need to be tagged with

two different types. We can't make the precondition of the Hoare triple any weaker; if either the is_valid_w32 or is_valid_w8 clauses are removed, one of the two guard statements in the function body will fail.

So how do these observations explain our apparently unsound abstraction? Recall that the definition of $\text{corres}_{\text{HA}}$ starts with the implication:

$$\forall s.\ \neg\ \text{failed}\ (A\ (st\ s)) \longrightarrow \ldots$$

That is, refinement only holds for states which, when abstracted, run to completion without failure. In the case of type_unsafe′, every abstracted state will *have* to fail one of the two guards in the function's body and hence, refinement doesn't hold.

In Section 5.4, we looked at how the abstractions of AutoCorres can be formally linked to the original concrete C programs. Carrying out this process on type_unsafe′ would have revealed that the generated abstract specification always leads to failure for states that came from our original Simpl program, and hence the generated abstract program is of little use.

7.4 Abstracting C structures

C structures are used in most non-trivial C programs, allowing new compound types to be formed by combining several simpler types. When the C-to-Isabelle parser translates a C structure, it generates (*i*) an Isabelle record representing the structure; (*ii*) a deeply embedded encoding of the byte layout of the C structure; and (*iii*) a deeply embedded encoding of how the byte layout of the structure corresponds to the Isabelle record.

For example, given the C structure:

```
struct node {
    struct node *next;
    int data;
};
```

the C-to-Isabelle parser would generate the following record:

record *node_C* =
  next_C :: *node_C ptr*
  data_C :: *sword32*

The deeply embedded encodings of the byte layout of the structure and its correspondence to the generated record were developed by Tuch et al., and described in detail in [100].

When the C-to-Isabelle parser translates field accesses such as `p->data` into Simpl, it generates pointer/offset expressions, such as:

$$p\ +_{\text{p}}\ \text{int}\ (\text{field\_offset}\ \text{TYPE}(\textit{node\_C})\ [\text{''data''}])$$

Here, the function field_offset decodes the structure layout information of the *node_C* type and returns the number of bytes between the beginning of the `node` structure and

Structure abstraction

$$\frac{\text{valid\_field } st\ f_n\ f_r\ f_w\ h_r\ h_w\ v_r\ v_w\ t\_hrs_r\ t\_hrs_w \quad \text{abs\_expr}_{\text{HA}}\ st\ P\ e\ e'}{\text{abs\_expr}_{\text{HA}}\ st\ (\lambda s.\ P\ s \wedge v_r\ s\ (e\ s))\ (\lambda s.\ f_r\ (h_r\ s\ (e\ s)))\ (\lambda s.\ \text{h\_val}\ (\text{hrs\_mem}\ (t\_hrs_r\ s))\ (\text{Ptr}\ \&(e'\ s{\rightarrow}f_n)))}\ \textsc{HeapAbsFieldRead}$$

$$\frac{\text{valid\_field } st\ f_n\ f_r\ f_w\ h_r\ h_w\ v_r\ v_w\ t\_hrs_r\ t\_hrs_w \quad \text{abs\_expr}_{\text{HA}}\ st\ P\ a\ a' \quad \text{abs\_expr}_{\text{HA}}\ st\ Q\ b\ b'}{\begin{array}{l}\text{abs\_modifies}_{\text{HA}}\ st\ (\lambda s.\ P\ s \wedge Q\ s \wedge v_r\ s\ (a\ s))\\ \quad (\lambda s.\ h_w\ (\lambda old.\ old(a\ s := f_w\ (b\ s)\ (old\ (a\ s))))\ s)\\ \quad (\lambda s.\ t\_hrs_w\ (\text{hrs\_mem\_update}\ (\text{heap\_update}\ (\text{Ptr}\ \&(a'\ s{\rightarrow}f_n))\ (b'\ s)))\ s)\end{array}}\ \textsc{HeapAbsFieldWrite}$$

$$\frac{\text{valid\_field } st\ f_n\ f_r\ f_w\ h_r\ h_w\ v_r\ v_w\ t\_hrs_r\ t\_hrs_w \quad \text{abs\_expr}_{\text{HA}}\ st\ P\ a\ a'}{\begin{array}{l}\text{abs\_expr}_{\text{HA}}\ st\ (\lambda s.\ P\ s \wedge v_r\ s\ (a\ s))\\ \quad (\lambda s.\ \text{True})\ (\lambda s.\ \text{c\_guard}\ (\text{Ptr}\ \&(a'\ s{\rightarrow}f_n)))\end{array}}\ \textsc{HeapAbsFieldGuard}$$

Table 7.4: Rules used by AutoCorres to abstract reads and writes to fields of structures.

the `data` field. In this example, the offset is 4. We use the notation $\&(p{\rightarrow}[''\text{data}''])$ as a shorthand for the above expression.

The last argument of field_offset takes a list of strings representing field names. For instance, `p->a.b.c` would be represented as:

$$\&(p{\rightarrow}[''\text{a}'',\ ''\text{b}'',\ ''\text{c}''])$$

Our goal when abstracting programs that use structures is to convert these pointer/offset expressions into record accesses. For instance, expressions such as:

$$\text{h\_val}\ (\text{hrs\_mem}\ (\text{t\_hrs\_}'\ s))\ (\text{Ptr}\ \&((p :: node\_C\ ptr){\rightarrow}[''\text{data\_C}'']))$$

should become a simple access to the generated *node_C* heap:

$$\text{data\_C}\ (\text{heap\_node\_C}\ s\ p)$$

where heap_node_C $s\ p$ returns a *node_C* record, and data_C accesses the `data` field of the record. We use the notation $s[p]{\rightarrow}\text{data}$ to represent an access of the field data on the heap associated with pointer p in state s, and the notation $s[p{\rightarrow}\text{data} := v]$ to denote an update of field data at pointer p in state s to the new value v.

To carry out the conversion, AutoCorres uses the deeply embedded structure information generated by the C-to-Isabelle parser to form a theorem that links the byte-level structure representation to the desired abstract representation. The theorem generated

by AutoCorres is a predicate named valid_field, with the following (mildly daunting) definition:

$$
\begin{aligned}
&\text{valid\_field } st\ f_n\ f_r\ f_w\ h_r\ h_w\ v_r\ v_w\ t\_hrs_r\ t\_hrs_{rw} \equiv \\
&\quad (\forall s\ p.\ v_r\ (st\ s)\ p \longrightarrow \\
&\qquad \text{h\_val } (\text{hrs\_mem } (t\_hrs_r\ s))\ (\text{Ptr } \&(p{\rightarrow}f_n)) = \\
&\qquad\quad f_r\ (h_r\ (st\ s)\ p)) \wedge \\
&\quad (\forall s\ p\ val. \\
&\qquad v_r\ (st\ s)\ p \longrightarrow \\
&\qquad\quad st\ (t\_hrs_{rw}\ (\text{hrs\_mem\_update } (\text{heap\_update } (\text{Ptr } \&(p{\rightarrow}f_n))\ val))\ s) = \\
&\qquad\quad\quad h_w\ (\lambda old.\ old(p := f_w\ val\ (old\ p)))\ (st\ s)) \wedge \\
&\quad (\forall s\ p.\ v_r\ (st\ s)\ p \longrightarrow \text{c\_guard } p) \wedge \\
&\quad (\forall p.\ \text{c\_guard } p \longrightarrow \text{c\_guard } (\text{Ptr } \&(p{\rightarrow}f_n)))
\end{aligned}
$$

The arguments f_n, f_r and f_w refer to a field's name, field reader and field writer, respectively. In our example above, these would be ''data'', data_C and data_C_update. The arguments h_r, h_w, v_r and v_w refer to the heap_x reader/writer and is_valid_x reader/writer for the structure (heap_node_C, heap_node_C_update, is_valid_node_C and is_valid_node_C_update in this example). Finally, the arguments st, t_hrs_r and t_hrs_w match those of valid_heap, referring to the state translation function and the reader and writer of the C-to-Isabelle parser's t_hrs_′ constant.

The valid_field predicate states four facts about the field:

- Assuming a heap location is valid, then accessing the bytes at offset f_n on the concrete heap is equivalent to accessing f_r of the abstract record;
- Similarly, writing to the bytes at offset f_n on the concrete heap is equivalent to writing to the field f_w of the abstract record;
- If the is_valid_x function v_r is true at pointer p, then c_guard p also holds for that pointer; and finally,
- If c_guard p holds for a pointer p, then it will also hold for the pointer into the struct Ptr $\&(p{\rightarrow}f_n)$.

Each of these items can be proved automatically for each field of every struct by AutoCorres, using information generated by the C-to-Isabelle parser about structures and structure layouts.

To actually carry out the abstraction, AutoCorres uses the rules listed in Table 7.4. While the shear amount of syntax in the rules makes them intimidating, nothing conceptually difficult is going on. HeapAbsFieldRead converts a read of a structure field into a read of the abstract record's field; HeapAbsFieldWrite does the equivalent operations for writes to structure fields; while HeapAbsFieldGuard converts a c_guard statement of a pointer directly accessing a structure's field into a check that the structure itself is valid. All of these rules can be simply derived from the definition of valid_field.

```
int suzuki(struct node *w, struct node *x,
       struct node *y, struct node *z)
{
   w->next = x; x->next = y; y->next = z; x->next = z;
   w->data = 1; x->data = 2; y->data = 3; z->data = 4;
   return w->next->next->data;
}
```

```
suzuki′ w x y z ≡
       do guard (λs. is_valid_node_C s w);
          modify (λs. s[w→next := x]);
          guard (λs. is_valid_node_C s x);
          modify (λs. s[x→next := y]);
          guard (λs. is_valid_node_C s y);
          modify (λs. s[y→next := z]);
          modify (λs. s[x→next := z]);
          modify (λs. s[w→data := 1]);
          modify (λs. s[x→data := 2]);
          modify (λs. s[y→data := 3]);
          guard (λs. is_valid_node_C s z);
          modify (λs. s[z→data := 4]);
          guard (λs. is_valid_node_C s s[w]→next);
          guard (λs. is_valid_node_C s s[s[w]→next]→next);
          gets (λs. s[s[s[w]→next]→next]→data)
       od
```

Figure 7.9: *Suzuki's challenge*, an example of deep nesting of pointer updates taken from Suzuki [97], along with AutoCorres' abstraction of the function.

7.4.1 Example: Suzuki's challenge

In Section 7.2.3, we described some limitations of Tuch's heap lifting approach in solving Suzuki's challenge [97]. In this example, we revisit the challenge using AutoCorres' heap abstraction.

The function and AutoCorres' abstraction is shown in Figure 7.9. While the output of AutoCorres appears long, each modify statement corresponds directly to a heap update in the source program.

We can prove that the function `suzuki` returns the value `4` with the Hoare triple:

$$\begin{array}{l}\{\,\lambda s.\ \text{distinct}\ [w,\ x,\ y,\ z]\ \wedge \\ \qquad (\forall p \in \{w,\ x,\ y,\ z\}.\ \text{is\_valid\_node\_C}\ s\ p)\,\} \\ \quad \text{suzuki}'\ w\ x\ y\ z \\ \{\,\lambda rv\ s.\ rv = 4\,\}\end{array}$$

That is, assuming that the pointers w, x, y and z are distinct and all valid, the return value of the function is 4. The proof proceeds by unfolding the definition of suzuki′, running the VCG, and then running Isabelle/HOL's `auto` tactic. The simplicity of discharging this goal is an encouraging indication that we have solved one of the scalability problems of Tuch's approach.

```
struct node *reverse(struct node *list) {
    struct node *rev = NULL;
    while (list) {
        struct node *next = list->next;
        list->next = rev; rev = list; list = next;
    }
    return rev;
}
```

Figure 7.10: A C implementation of a function that reverses a linked list in-place.

7.4.2 Example: in-place reversal of a linked list

Proving correctness of an in-place list reversal function is generally considered the *hello world* of pointer program verification. Such a function takes a singly linked list as an input, reverses the order of the nodes without allocating memory, and then returns a pointer to the beginning of the (now-reversed) list. A C implementation of such a program is shown in Figure 7.10.

Heap abstraction converts the program from using a byte-level heap, where the code construct `list->next` is represented as a pointer/offset pair, into an object-level heap consisting of *node_C* records. The two `reverse` functions—with heap abstraction disabled and with heap abstraction enabled—are shown in Figure 7.11.

We describe a full proof of the function's correctness in Section 8.1.1, but simply note here that heap abstraction allows us to specify the correctness result relatively easily:

$$\{\lambda s.\ \mathsf{list}\ s\ p\ xs\}\ \mathsf{reverse}'\ p\ \{\lambda rv\ s.\ \mathsf{list}\ s\ rv\ (\mathsf{rev}\ xs)\}$$

This Hoare triple states that, assuming the linked list xs is at location p, then the function will finally return a pointer to the reversed linked list rev xs. This result requires around 90 lines of proof: 70 lines of that are generic theorems about linked lists, while 18 lines are required for the actual proof of the function reverse′.

While the in-place linked list reverse program has been verified several times in various contexts [18, 23, 43, 70, 89, 91], attempting to carry out such reasoning directly on a byte-level heap would require a heroic effort. The only work we are aware of that has attempted such a feat is Tuch [100] who—not content to simply show that such a feat was possible—carried out the byte-level proof *twice*. The first was using the heap lifting framework describe in this chapter, while the second proof utilised a separation logic framework defined on the byte level heap. In both cases, non-trivial reasoning machinery was required to carry out the proof. In contrast, our work allows high-level reasoning on byte-level heaps without requiring the user to interact with such machinery.

7.5 Mixing low-level and high-level code

One of our original motivations for using the C programming language was its ability to interact with the heap at a low-level. Heap abstracted code, however, requires that

Byte-level heap

```
reverse_byte_heap' list ≡
    do (list, rev) ←
        whileLoop (λ(list, rev) a. list ≠ NULL)
            (λ(list, rev).
                do guard (λs. c_guard list);
                   next ←
                     gets (λs. h_val (hrs_mem (t_hrs_' s))
                                 (Ptr &(list→[''next_C'']))));
                   modify
                     (t_hrs_'_update
                         (hrs_mem_update
                             (heap_update (Ptr &(list→[''next_C'']))
                                 rev)));
                   return (next, list)
                od) (list, NULL);
       return rev
    od
```

Abstracted heap

```
    reverse' list ≡
        do (list, rev) ←
            whileLoop (λ(list, rev) a. list ≠ NULL)
                (λ(list, rev).
                    do guard (λs. is_valid_node_C s list);
                       next ← gets (λs. s[list]→next);
                       modify (λs. s[list→next := rev]);
                       return (next, list)
                    od) (list, NULL);
           return rev
        od
```

Figure 7.11: The `reverse` program listed Figure 7.10, translated using a byte-level heap (*top*) and with our abstracted heap (*bottom*).

memory is firmly tagged to being accessed only as a single type, which prevents type-unsafe functions such as `memcpy` from being used.

Our solution is to allow the user to indicate which functions should be abstracted and which should remain in the low-level memory model. The former has the benefits of simplified reasoning with heap abstraction, while the latter allows type-unsafe operations to be reasoned about.

Calls that take place from abstracted code into low-level code use the function exec_concrete M, where M is the low-level function to be executed. exec_concrete non-deterministically selects a low-level state corresponding to the current high-level state, executes the monad M, and then translates the resulting low-level state back into an abstracted state. It is defined as follows:

$$\begin{aligned}&\mathsf{exec\_concrete}\ st\ M \equiv \\ &\quad \lambda s.\ (\{(r,\ t).\ \exists s'\ t'.\ s = st\ s' \wedge t = st\ t' \wedge (r,\ t') \in \mathsf{results}\ (M\ s')\},\\ &\qquad \exists s'.\ s = st\ s' \wedge \mathsf{failed}\ (M\ s'))\end{aligned}$$

While the existentials in the definitions above may initially appear to be difficult to reason about, we can prove the following Hoare triple to reason about calls to exec_concrete, which avoids the existentials altogether:

$$\frac{\{\lambda s.\ P\ (st\ s)\}\ M\ \{\lambda r\ s.\ Q\ r\ (st\ s)\}}{\{P\}\ \mathsf{exec\_concrete}\ st\ M\ \{Q\}}$$

Functions that are marked to not use heap abstraction may need to call functions that *are* heap abstracted. We thus also create an analogous function exec_abstract allowing such calls from a function using a byte-level heap to a function using an abstracted heap. exec_abstract is defined as follows:

$$\begin{aligned}&\mathsf{exec\_abstract}\ st\ M \equiv \\ &\quad \lambda s'.\ (\{(r',\ t').\ \exists t.\ t = st\ t' \wedge (r',\ t) \in \mathsf{results}\ (M\ (st\ s'))\},\\ &\qquad \exists s.\ s = st\ s' \wedge \mathsf{failed}\ (M\ (st\ s')))\end{aligned}$$

The exec_abstract function abstracts the input state, executes the monad M on the abstract state, and then non-deterministically selects a concrete state corresponding to the output abstract states. We can use the following rule to reason about Hoare triples using exec_abstract:

$$\frac{\{P\}\ M\ \{\lambda r\ s.\ \forall t.\ st\ t = s \longrightarrow Q\ r\ t\}}{\{\lambda s.\ P\ (st\ s)\}\ \mathsf{exec\_abstract}\ st\ M\ \{Q\}}$$

The universal quantifier in the assumption's postcondition makes this rule less convenient to use than the Hoare-rule for exec_concrete. In our experience calls to exec_abstract are far less common than exec_concrete, however, because low-level type-unsafe operations that cannot be abstracted tend to be constrained to leaf functions.

7.5.1 Example: `memset`

The canonical example of a necessarily type-unsafe function in C is `memset`, shown in Figure 7.12. The function sets `n` bytes to the value `c`, starting from the pointer `dest`. The

```
void* memset(void *dest, int c, unsigned n)
{
    unsigned char *d = dest;
    while (n > 0) {
        *d = c;
        d++;
        n--;
    }
    return dest;
}

void zero_node(struct node *node)
{
    memset(node, 0, sizeof(struct node));
}
```

Figure 7.12: An implementation of the C type-unsafe function `memset`, which sets `n` bytes from the pointer `dest` to the value `c`; and a function `zero_node` that calls `memset` to zero out a `struct node`.

```
memset′ dest c n ≡
    do whileLoop (λ(d, n) a. 0 < n)
            (λ(d, n).
                do guard (λs. c_guard d);
                    modify (t_hrs_′_update
                                (hrs_mem_update (heap_update d (scast c))));
                    return (d +p 1, n − 1)
                od) (ptr_coerce dest, n);
        return dest
    od

zero_node′ node ≡
    do exec_concrete st (memset′ (ptr_coerce node) 0 (size_of TYPE(node_C)));
        skip
    od
```

Figure 7.13: The output of AutoCorres on the `memset` and `zero_node` functions. Heap abstraction is disabled on `memset`, so the C-to-Isabelle parser's byte-level heap remains in use on the function, while the call to `memset` in `zero_node` is wrapped in the exec_concrete function.

function is typically called with `c` set to `0` to zero out memory at a specified location. Figure 7.12 additionally shows a function `zero_node` that uses `memset` to zero an input `struct node *` object.

If we attempted to use heap abstraction on the function `memset`, AutoCorres would generate an output specification where `memset` zeroed the heap_w8 heap, but left all other heaps untouched. We would not be able to reason about calls to `memset` that attempted to modify any type other than *word8*. We thus mark `memset` as being unsuitable for heap abstraction when invoking AutoCorres; this gives the generated function shown in Figure 7.13.

We proceed to show the following Hoare triple about `memset`:

$$\begin{array}{l}\forall s_o.\ \{\lambda s.\ s = s_o \wedge n < 2^{32} \wedge 0 \notin \{\text{ptr\_val } p\ ..+\ n\}\} \\ \quad \text{memset}'\ p\ c\ n \\ \quad \{\lambda rv\ s.\ s = \text{t\_hrs\_}'\text{\_update} \\ \qquad\qquad (\text{hrs\_mem\_update} \\ \qquad\qquad\quad (\text{heap\_update\_list}\ (\text{ptr\_val}\ p)\ (\text{replicate}\ n\ (\text{scast}\ c)))) \\ \qquad\qquad s_o\}\end{array}$$

The function heap_update_list $p\ l\ s$ in the postcondition updates the byte-level heap s (with type $word32 \Rightarrow word8$), writing the list of bytes l to location p. The function replicate $n\ c$ produces a list of length n containing the element c repeated. The precondition assumes that the size of memory `n` being zeroed is less than 2^{32} and that `NULL` does not fall in the address range being zeroed. If so, calling `memset` will result in `n` bytes starting from location `p` to be updated to the value `c`; the final state s will otherwise be unchanged. The proof proceeds by unfolding the definition of memset′, annotating the function's loop with an appropriate invariant, applying the VCG and then carrying out reasoning to show that several individual writes to bytes in memory are equivalent to a single larger heap_update_list call.

Despite the fact that our proof of `memset` takes place on the byte-level heap, we can still carry out a proof of zero_node′ using an abstracted heap. We start by proving a lemma that writing a list of zeroed bytes to a pointer location p is equivalent to writing a `struct node` to that same location:

$$\begin{array}{l}\text{heap\_update\_list}\ p\ [0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0] = \\ \qquad \text{heap\_update}\ (\text{Ptr}\ p)\ (\!|\ \text{next\_C} = \text{NULL},\ \text{data\_C} = 0\ |\!)\end{array}$$

This proof proceeds by appealing to the generated theorems about the layout of `struct node` in memory. These theorems are automatically generated by the C-to-Isabelle parser during its initial conversion from C to Isabelle/HOL.

With a byte-level proof about `memset` in hand, we can proceed to carry out a high-level proof on zero_node′. Our final theorem about the function is the Hoare triple:

$$\begin{array}{l}\forall s_o.\ \{\lambda s.\ \text{is\_valid\_node\_C}\ s\ p \wedge s = s_o\} \\ \quad \text{zero\_node}'\ p \\ \quad \{\lambda rv\ s.\ s = s_o[p := (\!|\ \text{next\_C} = \text{NULL},\ \text{data\_C} = 0\ |\!)]\}\end{array}$$

This states that, assuming that p is a pointer to any valid *node_C* object then, after zero_node′ executes, p will now point to a zeroed node_C while the rest of the state will remain unchanged.

The proof takes place by unfolding the definition of zero_node′, and then running the VCG (which includes our memset′ rule proven above) over the result. Three subgoals remain:

$$
\begin{array}{l}
1.\ \bigwedge s_o\ s.\ \mathsf{is\_valid\_node\_C}\ (\mathsf{st}\ s)\ p \Longrightarrow \mathsf{size\_of}\ \mathsf{TYPE}(\mathit{node\_C}) < 2^{32} \\
2.\ \bigwedge s_o\ s.\ \mathsf{is\_valid\_node\_C}\ (\mathsf{st}\ s)\ p \Longrightarrow \\
\qquad 0 \notin \{\mathsf{ptr\_val}\ (\mathsf{ptr\_coerce}\ p)\ ..{+}\ \mathsf{size\_of}\ \mathsf{TYPE}(\mathit{node\_C})\} \\
3.\ \bigwedge s_o\ s\ \mathit{rv}. \\
\quad \mathsf{is\_valid\_node\_C}\ (\mathsf{st}\ s)\ p \Longrightarrow \\
\quad \mathsf{st}\ (\mathsf{t\_hrs\_'\_update} \\
\qquad (\mathsf{hrs\_mem\_update} \\
\qquad\quad (\mathsf{heap\_update\_list}\ (\mathsf{ptr\_val}\ (\mathsf{ptr\_coerce}\ p)) \\
\qquad\qquad (\mathsf{replicate}\ (\mathsf{size\_of}\ \mathsf{TYPE}(\mathit{node\_C}))\ (\mathsf{scast}\ 0)))) \\
\qquad s) = \\
\quad (\mathsf{st}\ s)[p := (\!|\ \mathsf{next\_C} = \mathsf{NULL},\ \mathsf{data\_C} = 0\ |\!)]
\end{array}
$$

The first two subgoals originate from the precondition of memset′. The first subgoal states that the size of a *node_C* object is less than 2^{32}, which is trivially solved by the simplifier. The second states that `NULL` does not fall in the set of addresses occupied by the target *node_C*; this is proven by observing that the *node_C* pointed to by *p* is valid, and our state translation function ensures that the abstract heap only contains valid object if they are non-`NULL` on the concrete heap. Finally, the third subgoal requires us to show that writing zero bytes to the concrete heap is equivalent to writing a zero-*node_C* on the abstract heap, and is discharged by using our helper lemma above and appealing to the definition of st.

The proof all up requires approximately 100 lines: the bulk of the proof is 77 lines for the byte-level `memset` correctness proof, 13 lines are required to prove the encoding of the empty *node_C* is equivalent to eight zero bytes, and 11 lines are required for the final `zero_node` proof.

7.6 Related work

Our work is heavily inspired by that of Tuch et al. [100–103], who developed the heap lifting framework that our own work is based on. Tuch et al.'s framework allows higher-level reasoning on a byte-level heap by providing users with mechanisms to manually interpret low-level operations as operations on a Burstall-Bornat style heap. Our contribution is to automate this process; instead of providing the user with tools for higher-level reasoning, we rewrite the program being reasoned on to directly operate on a Burstall-Bornat style heap. Additionally, we developed a new, simplified implementation of Tuch et al.'s heap lifting framework more suitable for mechanised reasoning. Our work uses this framework internally so that a formal connection between the input byte-level specification and the output high-level specification exists.

Tuch et al.'s original framework supports reasoning about programs that dereference pointers to fields of structures. Our simplified version of Tuch et al.'s work and method of program rewriting cannot deal with this particular case at this time. The end-user,

however, always has the ability to disable heap abstraction and reason using Tuch et al.'s manual framework directly. Tuch et al.'s original framework additionally implements separation logic on the byte-level heap, which our own implementation does not. We feel that the better approach is to separate the concerns: apply heap abstraction first, and then apply reasoning using separation logic on the abstract heap. For instance, instantiating Klein et al.'s separation algebra [58] on AutoCorres' output can be carried out with relatively little effort.

Other work that models the C heap at the byte level includes the verified CompCert C semantics [67], and Ellison and Roşu's C semantics [42]. This latter work defines a highly complete and validated C semantics with a memory model that is essentially a map to blocks of bytes.[11] To our knowledge, no mechanisms for higher-level heap reasoning have yet been developed for these models, though we believe our work could be applied to enable Burstall-Bornat style reasoning.

Gast [45] developed a framework for reasoning about byte-level heaps by allowing the memory layout (and in particular, how objects overlap) to be cleanly specified in the logic. Using this framework, Gast was able to verify a C implementation of the challenging Schorr-Waite algorithm. Again, the goal of our work is not to provide users with powerful frameworks for low-level reasoning but to abstract specifications, avoiding the need for such frameworks. Like Tuch et al.'s framework, Gast's framework could be directly applied in the limited cases where heap abstraction cannot be used, such as when reasoning about type-unsafe code.

Existing C verification frameworks that provide the user with an abstract view of the system heap tend to do so axiomatically. Moy's translation of C into the Frama-C framework with the Jessie plugin, for instance [72], provides an axiomatic typed-heap memory model with limited support for pointer casts. The SMT-based VCC tool [31], originally with an untyped memory model, also supports a typed semantics, but the abstraction is justified by a pen and paper argument only. In contrast, our work provides a formal link down to the C byte-level heap; additionally, this link allows us to mix high-level reasoning and byte-level reasoning on demand.

To summarise, the key differentiator of our heap abstraction work is that (*i*) it allows reasoning at *both* a typed-heap model and byte level; (*ii*) that the lifting is *automatic*; (*iii*) the tool supports *any* C Standard retyping/casting operations, as long as the final pointer dereference is type-safe; and (*iv*) it provides an LCF-style foundational proof.

7.7 Conclusion

The heap abstraction process described in this chapter allows users to carry out high-level reasoning about programs that were initially modelled using a byte-level heap.

[11] This is closer to the C standard than our base byte-level model generated by Norrish's C-to-Isabelle parser, but verification relying on the standard alone is insufficient since it is routinely violated on purpose in systems code [57].

Heap abstraction occurs automatically, and can take place on type-safe portions of the user's program, without sacrificing the ability for users to drop back into the underlying byte-level model where necessary.

In this chapter we presented some simple examples demonstrating how heap abstraction simplifies reasoning. In the next chapter we show some more substantial examples, looking at how all of the abstraction steps we have described so far in this document can be combined to carry out high-level reasoning about non-trivial algorithms.

Chapter Summary

- Conservative translations of C programs represent memory using a byte-level model. Such byte-level models of memory are difficult to reason about, due to inter-type aliasing, the potential for partially overlapping objects, alignment constraints imposed by C, and so on.
- We implemented a simplified version of Tuch et al.'s heap lifting framework, which allows users to manually translate low-level heap operations into higher-level equivalents.
- Using our simplification of Tuch et al.'s model, we then developed techniques to *automatically* and *verifiably* abstract program specifications to operate on a Burstall-Bornat style heap. Our work retains a formal connection down to the original byte-level heap, and also allows mixing low-level reasoning and high-level reasoning on a per-function basis.

8 Evaluation and experience

For AutoCorres to be practical in real-world problems, it not only needs to generate *useful* abstractions of low-level C code, but must also be *scalable* to problems of the size actually seen in industry. All of the examples we have presented thus far have been relatively simple, making it hard to establish if we have achieved either of these two goals. In this chapter, we attempt to address these issues.

To evaluate how well AutoCorres abstracts C code, we take an existing high-level proof [70] of the Schorr-Waite graph marking algorithm and determine the difficulty of porting the high-level proof to a concrete C implementation with the help of AutoCorres. We find that the original high-level proof—written over a decade ago by unrelated authors—ports relatively easily to the output of AutoCorres, suggesting that reasoning at a relatively abstract level about concrete C code is possible. The details and results are in Section 8.1.

Additionally, we present a bigger picture overview of areas were AutoCorres is currently being used by others in larger proof projects, including an external evaluation of the tool. While much of work using AutoCorres is still in its early stages, initial results suggest that AutoCorres is able to scale to tens of thousands of lines of code, as required by real-world projects. We also show that AutoCorres is able to handle a sufficiently large subset of C to allow even complex programs, such as operating system kernels, to be processed. These results are presented in Section 8.2.

This chapter is based on the published work by Greenaway et al. [50], *Don't sweat the small stuff: formal verification of C code without the pain* in PLDI 2014. The work of porting Mehta and Nipkow's high-level proofs to C implementations was carried out by Japheth Lim, under the supervision of the author.

8.1 High-level reasoning with AutoCorres

The Schorr-Waite algorithm, Richard Bornat famously argued, is "the first mountain that any formalism for pointer aliasing should climb." [18]. His advice has not gone unheeded, with many papers demonstrating new program verification techniques on the algorithm: Hubert and Marché, for instance, previously verified a concrete C implementation using the Caduceus verification condition generator and the Coq theorem prover [54]. Earlier still, Mehta and Nipkow [70] verified the algorithm on a simple high-level imperative language in Isabelle/HOL, producing a readable machine-checked proof.

From our perspective, the latter proof is interesting because it verifies the Schorr-Waite algorithm on an idealised imperative language: the heap uses a Burstall-Bornat split heap memory model [18]; there is no concept of invalid pointers, such as unaligned pointers or unmapped memory; and the address space is infinite. All of these assumptions fail to hold on a low-level language such as C. A useful benchmark, then, is to determine if we can

1. Implement Mehta and Nipkow's version of the algorithm in plain C;
2. Use AutoCorres to automatically abstract the program; and then
3. Apply Mehta and Nipkow's existing proofs—written nearly a decade before AutoCorres was even conceived, and carried out on a very abstract heap—to the result.

We do not expect Mehta and Nipkow's proof to apply unchanged, but the goal is for any changes to be minimal.

In the next sections, we start by describing how Mehta and Nipkow's much simpler list reversal proof could be ported to a C implementation, in order to explain our approach, and then move on to describe how the more complex Schorr-Waite algorithm could be ported to C. The proof porting process was not carried out by the primary author of this work, but by an undergraduate student, Japheth Lim, under supervision.

8.1.1 In-place list reversal

As mentioned in Section 7.4.2, in-place linked list reversal has somewhat become the *hello world* of pointer aliasing programs. The list reversal function takes a singly linked list, destructively modifies it so that the order of the nodes is reversed, and returns a pointer to the head of the new list. The goal is to prove that the list returned by the function truly is the reverse of the input list. Mehta and Nipkow also used this example as an introductory exercise in their work [70], so it serves as a good example for demonstrating the differences between their program representations and those of AutoCorres.

To carry out their proof, Mehta and Nipkow developed a simple while language, including while-loops and conditionals. References in the language use the type

$$\textbf{datatype}\ 'a\ ref\ =\ \mathsf{Null}\ |\ \mathsf{Ref}\ nat$$

That is, an *'a ref* can be either Null, representing a null pointer; or Ref *addr*, representing a pointer to a valid object. The type variable *'a* is a phantom type variable, used simply to indicate the type of the pointer.

The language doesn't explicitly model a heap; instead, a heap is simulated by having global variables of type

$$'a\ ref\ \Rightarrow\ 'a$$

which map pointers to values. Such heaps are modified by assigning a new value to the variable, typically with only a single address updated. There is one such heap variable for each record field and type in the program.

For in-place list reversal, Mehta and Nipkow's algorithm and proof statement is as follows:

```
{List next p Ps}
    q := Null;
    WHILE p ≠ Null INV {...} DO
        t := p;
        p := next (addr p);
        next := next(t → q);
        q := t
    OD
{List next q (rev Ps)}
```

The theorem states that if the pointer *p* points to a linked list in the heap *next* containing the elements *Ps* then, at the end of the function, *p* will point to a linked list containing elements of the list *Ps* reversed. While not a complete specification of the function—it fails to mention what happens to nodes outside of the input list, for example—it does provide us with at least *some* confidence that the function successfully reverses the list.

The predicate List *next p Ps* in this theorem has type

$$('a\ \Rightarrow\ 'a\ ref)\ \Rightarrow\ 'a\ ref\ \Rightarrow\ 'a\ list\ \Rightarrow\ bool$$

and indicates that there is a valid linked list in the heap *next* starting from address *p*. The list *Ps* contains the pointers of every node in the linked list, and has a definition equivalent to

$$\mathsf{List}\ h\ p\ [\ ]\ =\ (p\ =\ \mathsf{Null})$$
$$\mathsf{List}\ h\ p\ (x{\cdot}xs)\ =\ (p\ =\ \mathsf{Ref}\ x\ \wedge\ \mathsf{List}\ h\ (h\ x)\ xs)$$

Mehta and Nipkow's proof proceeds by first building a library of theorems about the behaviour of the List predicate, providing an invariant for the while loop, running a VCG on the statement, and finally discharging the goals using standard Isabelle/HOL tactics. The bulk of the work required to verify the reverse function is in developing the library of list theorems, with just a few lines of proof required to verify the reverse function itself.

```
struct node *reverse(struct node *list) {
    struct node *rev = NULL;
    while (list) {
        struct node *next = list->next;
        list->next = rev; rev = list; list = next;
    }
    return rev;
}
```

```
reverse′ list ≡
    do (list, rev) ←
        whileLoop (λ(list, rev) a. list ≠ NULL)
            (λ(list, rev).
                do guard (λs. is_valid_node_C s list);
                   next ← gets (λs. s[list]→next);
                   modify (λs. s[list→next := rev]);
                   return (next, list)
                od) (list, NULL);
        return rev
    od
```

Figure 8.1: ANSI C implementation of in-place linked list reversal, and its translation into Isabelle/HOL by AutoCorres.

Porting the list reversal proof to AutoCorres

Our C implementation of Mehta and Nipkow's list reversal algorithm, together with the corresponding output of AutoCorres, is shown in Figure 8.1. To port Mehta and Nipkow's proof so that it applies to the output of AutoCorres, we had to resolve the following three differences:

- The original proof uses the *'a ref* datatype to distinguish between the null pointer (Null) and pointers to valid objects (Ref *addr*), while C uses the `NULL` sentinel value. For example, the above List predicate would need to be modified to

  $$\text{List } h\ p\ [] = (p = \text{NULL})$$
  $$\text{List } h\ p\ (x{\cdot}xs) = (p \neq \text{NULL} \wedge p = x \wedge \text{List } h\ (h\ p)\ xs)$$

  which explicitly checks that the constant NULL does not appear mid-list, instead of exploiting the type system to ensure its absence. This seemingly trivial difference in the types of pointers necessitated tweaks in the majority of the list definitions and proof statements; despite this, once the base definitions were updated we could use the original proof scripts mostly unchanged.

- In Mehta and Nipkow's language model there is no concept of an invalid heap address. In contrast, the output of AutoCorres contains guard statements to ensure that each pointer access is valid. We further modified the definition of List shown above to additionally assert that all elements in the list are valid pointers; this was enough to automatically discharge the guards in the final proof.

```
{R = reachable (relS {l, r}) {root} ∧ (∀x. ¬ m x) ∧ iR = r ∧ iL = l}
  t := root; p := Null;
  WHILE p ≠ Null ∨ t ≠ Null ∧ ¬ t^.m INV {...} DO
      IF t = Null ∨ t^.m THEN
          IF p^.c THEN
              q := t; t := p; p := p^.r; t^.r := q
          ELSE
              q := t; t := p^.r; p^.r := p^.l; p^.l := q; p^.c := True
          FI
      ELSE
          q := p; p := t; t := t^.l; p^.l := q; p^.m := True; p^.c := False
      FI
  OD
{(∀x. (x ∈ R) = m x) ∧ r = iR ∧ l = iL}
```

Figure 8.2: Mehta and Nipkow's correctness statement of the Schorr-Waite algorithm, reproduced from [70].

- The original proof shows partial correctness, while AutoCorres requires total correctness for its automatically generated refinement theorem to hold. We extended Mehta and Nipkow's proof to include a termination argument; in particular, we showed that the size of the list yet to be reversed decreases each loop iteration.

With these adjustments, we completed the same main proof of correctness using the same loop invariant as Mehta and Nipkow. Overall, only minor effort was required to obtain a C-level correctness guarantee from a high-level algorithmic proof. While in this case it may have been easier to prove the algorithm from scratch, this approach used to modify the Mehta and Nipkow proofs can be carried over to the much more complex Schorr-Waite algorithm proof, described below.

8.1.2 Schorr-Waite algorithm

The Schorr-Waite algorithm [94] enumerates all nodes in a graph. While such a problem can be trivially solved using a stack linear in the size of the graph, the Schorr-Waite algorithm requires only two bits of storage per node. The algorithm achieves this by reversing the pointers that link nodes in the graph so that the algorithm is able to backtrack, and then later restoring the pointers so that the input graph is back in its original form by the end of the algorithm. Because of its low memory requirements, the Schorr-Waite algorithm was originally proposed as the core element of a mark-and-sweep garbage collector; graph nodes not reached by the algorithm are no longer live, so can be reallocated. For brevity, we do not attempt to describe the algorithm in detail, but instead refer interested readers to one of the many descriptions available [54, 70, 94].

We based our C implementation, shown in Figure 8.3, directly on the high-level imperative implementation of Mehta and Nipkow, reproduced in Figure 8.2. Each graph

```
struct node {
    struct node *l, *r;
    unsigned m, c;
};

void schorr_waite(struct node *root) {
    struct node *t = root, *p = NULL, *q;
    while (p != NULL || (t != NULL && !t->m)) {
        if (t == NULL || t->m) {
            if (p->c) {
                q = t; t = p; p = p->r; t->r = q;
            } else {
                q = t; t = p->r; p->r = p->l;
                p->l = q; p->c = 1;
            }
        } else {
            q = p; p = t; t = t->l; p->l = q;
            p->m = 1; p->c = 0;
        }
    }
}
```

Figure 8.3: Our implementation of the Schorr-Waite algorithm in C, based on Mehta and Nipkow's high-level implementation.

node contains two pointers `l` and `r` pointing to the *left* and *right* child, respectively. Additionally, each graph node contains two bits which are required by the algorithm. The *marked* bit `m` is set when a node has been visited by the algorithm. When the algorithm completes, all nodes reachable from the root will have their marked bit set. The *child* bit `c` is used to track which children of the current node have already been visited.

Mehta and Nipkow's proof states that, assuming (*i*) R is the set of nodes reachable from the root of the graph *root*, and (*ii*) No nodes in memory are already marked; then, after the algorithm finishes (*i*) Every node will be marked if and only if it is in the set R; and (*ii*) The pointers of all nodes will match what they started as (that is, the heap r will equal its initial value iR, and the heap l will also match its initial value iL).

The implementation uses the same while-language as in the previous example, but with the additional syntax $a\hat{}.f$ for accessing the heap f at location a and $a := f\hat{}.v$ for updating the heap f at location a to value v.

Porting the Schorr-Waite proof to AutoCorres

After translating our C implementation of Schorr-Waite using AutoCorres, we reused Mehta and Nipkow's existing proof script to verify the algorithm. This reuse presented the same set of differences as in the list reversal problem:

- Again, replacing the *'a ref* type with the `NULL` sentinel used by C caused no significant semantic changes to the proof, but did require updates to the base definitions.

Table 8.1: A comparison of the lines of proof required for our work, for Mehta and Nipkow's proof [70] in Isabelle/HOL (*M/N*), and Hubert and Marché's proof [54] in Coq (*H/M*).

Component	***This Work***	**M/N**	**H/M**
List definitions	64	62	$\sim$ 900
Partial correctness	528	489	$\sim$ 1 400
Fault freedom	44	—	
Termination	160	—	$\sim$ 900
Miscellaneous	11	26	—
Total	807	577	3317

- Since Mehta and Nipkow's language model has no concept of invalid addresses, *it is not possible* to construct an invalid graph, as every possible state of memory forms a valid input graph. There is no such luxury when using C, so we needed to add a new precondition that all nodes in the set of reachable addresses R are valid. We additionally needed to introduce a new loop invariant to the main loop that asserts this fact.
- Again, Mehta and Nipkow's proof is a partial correctness result, while we required total correctness. We modified the proof to include a new termination argument, requiring around 160 lines of new proof script. In particular, we annotated the main loop body with the measure used in Bornat's proof of the Schorr-Waite algorithm [18], and showed that it decreases.

Overall, while we needed to make several changes to the base definitions of Mehta and Nipkow's proof script before we could apply it to the output of AutoCorres, most of these changes were simple; the main body of the original proof could be used relatively unmodified. Table 8.1 shows the number of lines required for our modified proof and for Mehta and Nipkow's original. The list definitions shared 48 lines (76%) while the main body of the proof shared 335 lines (66%). Of the remaining lines, the majority of changes were differences in syntax between the two verification frameworks, and due to minor differences in the output of the VCG tools.

Table 8.1 also includes numbers from Hubert and Marché's earlier Coq proof of the Schorr-Waite algorithm in C [54], which has a C implementation semantically equivalent to our own. Lines of proof script are not directly comparable between the two provers: Isabelle/HOL tends to provide more automation than Coq (often resulting in smaller proof scripts); however, Mehta and Nipkow's proof was intended to be highly readable (leading to a longer, more verbose proof script). Generously assuming Isabelle to be twice as concise as Coq, the size reduction compared to the previous C verification is striking.

Overall, in both the linked list reversal program and Schorr-Waite implementation, we were able to apply an existing proof of an abstract algorithm almost directly to our automatically produced abstraction of a low-level C implementation. Even without taking into account the fact that we proved stronger statements than the original, the size increase was moderate at most, and more importantly, the proof complexity

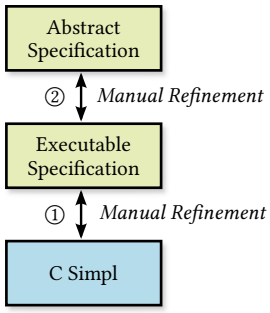


Figure 8.4: The structure of the seL4 refinement proof. The proof is split into two phases: a proof of refinement between the C and the executable specification; and a proof between the executable specification and the abstract specification.

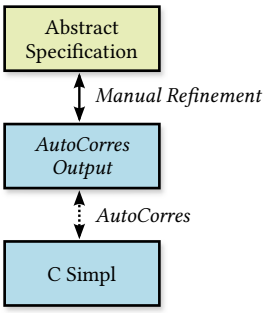


Figure 8.5: An alternative proof design for the seL4 proof using AutoCorres. The executable specification is replaced with the output of AutoCorres, and the first manual refinement proof eliminated.

remained similar to the original with unchanged or only minimally adjusted invariants. This strongly suggests that C verification can be performed from first principles at a comfortable level of abstraction.

8.2 Automatic abstraction in the large

While the examples shown in this document are relatively small, AutoCorres is designed to scale to non-trivial applications. In this section, we describe several past and ongoing verification projects using AutoCorres.

seL4 Microkernel The seL4 microkernel [57, 81, 82] is a formally verified operating system kernel. The proof shows that the C implementation of the kernel is a refinement of a hand-written high-level functional specification, named the *abstract specification*. While AutoCorres is not used in the seL4 proof at this stage, seL4 provided the inspiration for the tool, and hence also acts as a benchmark for testing the scalability and output of AutoCorres.

In particular, the seL4 refinement proof is broken into two main pieces: the first piece shows that the C implementation of seL4, translated by Norrish's C-to-Isabelle parser, is a refinement of an intermediate representation known as the *executable specification*. The second part of the proof shows that this executable specification is a refinement of the original high-level abstract specification. The executable specification acts as a 'stepping stone' between the concrete C code and the abstract specification, allowing low-level refinements involving details of C to be reasoned about separately from the higher-level refinements involving details about data structures and algorithms. The two pieces of the proof are then combined to formally link the C code to the abstract specification. The structure of the proof is depicted in Figure 8.4.

Both the low-level proof (i.e., the proof that links the C code to the executable specification) and the high-level proof (i.e., the proof that links the executable specification to the abstract specification) are manually written Isabelle/HOL proofs. While the high-level proof required some level of creativity to initially craft, the low-level proof that connects the C to the executable specification was far more mechanical in nature. Despite this, the low-level proof still required three person-years of effort to carry out.

AutoCorres was developed with the goal of automating the low-level phase of the proof chain in future projects. It automatically generates an intermediate executable specification directly from the C code, along with a formal proof that the C implementation refines this generated specification. While the high-level proof of refinement would remain a manual step (taking place between the output of AutoCorres and the abstract specification), by automating the low-level proof, the total effort would be significantly reduced.

To be practical for a future project similar to seL4, AutoCorres needs to be able to scale up to the size of seL4, and also be able to handle all of the complexities of C present in the seL4 source code. Our results in Table 8.2 and Table 8.3 show that AutoCorres has indeed been able to meet this requirement.

eChronos Real Time Operating System eChronos [80] is a high-assurance real-time operating system for small microcontrollers without memory protection. It consists of around 600 lines of C code, and provides system primitives such as threads, mutexes, signals and interrupt handling.

Like the seL4 microkernel, eChronos has a high-level abstract specification written in Isabelle/HOL that describes the desired behaviour of the operating system. Also like seL4, the goal is to show that the C implementation of eChronos implements this high-level specification. The proof is structured using AutoCorres as the starting point, as described in the previous section and depicted in Figure 8.5. In particular, AutoCorres is used to abstract the C implementation of eChronos; this output is then manually shown to be a refinement of the original hand-written abstract specification.

At the time of writing, the proof is still in its early days with only around 40% of the functions in eChronos having been formally proven to be refinements of their abstract equivalents. Experience so far suggests that directly carrying out a proof of refinement between the output of AutoCorres and a high-level specification is feasible. Statistics about the output of AutoCorres when applied to the eChronos source code are presented in Table 8.2 and Table 8.3.

Graph Algorithm Verification Noschinski, Rizkallah and Mehlhorn [86] have used an early version of AutoCorres[1] to formally verify checkers for the widely used LEDA graph algorithms library [69].

In particular, untrusted C programs built using the LEDA libraries process a graph to determine whether it satisfies a particular property, such as if the graph is connected

[1] Noschinski et al. used a version of AutoCorres prior to heap abstraction or word abstraction being implemented.

or if the graph is planar. Once such an untrusted program has determined an answer, it also emits a *certificate* which provides evidence that the answer is correct. This certificate is then read by a second *checker* program, which performs the simpler task of ensuring that the certificate is valid, and hence the original program's answer was correct. This allows a high degree of confidence to be placed in the answer of the first program's answer by formally verifying the much simpler checker.

In their work, Noschinski et al. compared three methods of verifying the checker: (*i*) using a hand-written implementation of the checker in Simpl, based on the algorithms used in the C code, but not formally linked to it; (*ii*) using the output of AutoCorres; and (*iii*) an approach taken by previous work [1], which used the automatic code verifier VCC [31] to generate verification conditions that were then imported into Isabelle/HOL using HOL-Boogie [16], where they could then be solved.

Noschinski et al. observed that when verifying the graph algorithms using an early version of AutoCorres, the "effort for the verification of the C-version of the connectedness checker was about the same as in the VCC approach". Each approach had both advantages and disadvantages:

- VCC, based on the automatic theorem prover Z3, was better at solving low-level proof obligations about the C code without any user interaction. For instance, proof obligations showing that certain word arithmetic operations didn't overflow could be automatically discharged, while these had to be manually reasoned about using AutoCorres;
- The VCC approach required that certain details about the algorithms, such as the definitions of graphs and paths, were formalised in both Isabelle/HOL and VCC's weaker logic. As the algorithms being verified became more sophisticated, the burden of using VCC grew; and
- VCC—a large, complex program—requires a greater degree of trust to be placed in it to believe its results. In comparison, AutoCorres, developed in Isabelle/HOL, need not be trusted as its results are verified by the Isabelle/HOL kernel. Using both VCC and Isabelle/HOL additionally required an awkward step of importing VCC proof obligations into Isabelle/HOL, which could be avoided by committing to just a single reasoning platform.

Noschinski et al. additionally noted that reasoning about AutoCorres' output required more effort than reasoning on the hand-written abstract algorithm; this is not unexpected, as AutoCorres is lifting from a concrete C implementation. The biggest difficulties reported by the authors were in word proofs and having to manually abstract datatypes in the heap to abstract equivalents. Our hope is that our subsequent word abstraction and heap abstraction additions to AutoCorres (described in Chapter 6 and Chapter 7, respectively) will reduce this effort in future verification projects.

CapDL System Initialiser Complex safety- or security-critical systems frequently require several interconnected components to function. Bootstrapping a system from its initial state to a correctly configured system—which requires allocating resources to processes, setting up communication channels, setting up process address spaces and so

on—is a non-trivial task. Further, if a mistake is made in this configuration, it has the ability to undermine the security or safety of the system.

The CapDL SysInit tool [20] is a C program that runs on seL4 that solves the problem of boot-time system configuration. The tool is given a specification of a system—including details such as the number of components, the resources required by each component, and how components should be able to communicate among themselves—and initialises the seL4 system to match the specification. CapDL SysInit is unique in that it has a high-level specification formally proven to set up the specified system correctly. The proof has additionally been formally connected to the seL4 microkernel proof, which gives a final proof statement that both (*i*) the CapDL SysInit tool invokes the seL4 microkernel in the correct manner, so that (*ii*) the final kernel state matches the system initially requested by the user.

At the time of writing, there is no formal connection between the abstract specification of CapDL SysInit and its C implementation. AutoCorres is currently being evaluated in this project as the first step in a refinement proof between the C and the abstract specifications, similar to the structure of the eChronos proof described above. The hope is that AutoCorres will save a significant amount of human effort by automating the low-level aspects of the refinement proof. Statistics on the output of the CapDL System Initialiser produced by AutoCorres are shown in Table 8.2 and Table 8.3.

Piccolo Microkernel The Piccolo kernel is a prototype separation kernel designed for static systems with memory protection, developed internally at NICTA by Greenaway. It provides threads, synchronisation primitives, address spaces and inter-process communication. As with the seL4 kernel C implementation, AutoCorres can successfully process all of the low-level C code required by the kernel's implementation.

AutoCorres is currently being used to show that the Piccolo in-kernel memory manager is correct. For these proofs, the output of AutoCorres is being directly reasoned about; in particular, the output of the abstract heap-model provided by AutoCorres is being combined with an existing separation logic toolkit available on Isabelle/HOL [58, 59], with the goal of producing proof statements that can be readily composed with other proofs about the Piccolo kernel.

Student Usability Study The University of New South Wales runs a course named *COMP4161 -- Advanced Topics in Program Verification*. The course trains students in the use of the Isabelle/HOL interactive theorem prover, starting at basic lambda calculus and moving to training students to think about topics in software verification, such as Hoare-logic, loop invariants, etc. Students in the course are a mix of late-year undergraduate and early-year postgraduate students in computing degrees. The course is typically quite small, with only 8 students enrolled in 2013.

We asked students in the course to participate in a usability experiment to determine if AutoCorres improved productivity of C verification.[2] In particular, the final

[2] All participants gave informed consent to take part in this investigation, and the investigation was carried out with ethics approval from the University of New South Wales' Human Research Ethics Advisory Panel (approval *HREA 08/2013/47*).

exam of the course involved two questions that asked students to reason about C code. For one question, students reasoned using the output of Norrish's C-to-Isabelle parser directly, while for the other question, students used the output of AutoCorres.

The experiment was run as a 2×2 cross-over study. The class of 8 students was randomly divided into two equal-sized groups, which we name A/B and B/A:

- Group A/B was required to solve question (1) using AutoCorres, and question (2) using Norrish's C-to-Isabelle parser directly;
- Group B/A was required to solve question (1) using Norrish's C-to-Isabelle parser, and question (2) using AutoCorres.

That is, each of the two groups each solved one question using AutoCorres and one question using the C parser, with group B/A using the opposite tool of group A/B for each question.

Each student's assignment was marked, with marks given for correctly specifying a precondition, postcondition, and loop invariant for the program, as well as marks for successfully discharging the proof obligations generated by each tool's VCG. Our hypothesis was that questions completed using AutoCorres would, on average, score higher than those completed directly using Norrish's C-to-Isabelle parser.

A paired-samples, two-tailed t-test was conducted to compare the student's exam marks using the output of AutoCorres and using the output of the C-to-Isabelle parser directly. There was not a significant difference in exam marks for the question where students used AutoCorres ($M = 28.9$, $SD = 9.22$) and where the students used the C-to-Isabelle parser directly ($M = 23.5$, $SD = 13.9$); $t(7) = 1.93, p = 0.095$. The 95% confidence interval for the effect of using AutoCorres on the final assignment mark is between -1.2 and 12.0.

We believe the negative result of the experiment is due to the small sample size ($n = 8$), limiting the power of our test. For more conclusive results, we would need to complete the study with a larger sample size. We are, however, encouraged by the anecdotal observation that every student performed equal or better in their AutoCorres exam question than in their C-to-Isabelle parser exam question.

8.2.1 Summary and statistics of projects using AutoCorres

In summary, AutoCorres is being actively used or trialled in these projects in the following ways:

- The eChronos and CapDL SysInit tool use AutoCorres as a stepping stone in a larger, manual proof of refinement between an abstract program specification and the concrete C implementation. The end goal of these proofs are to show full functional correctness;
- The LEDA graph checkers similarly use AutoCorres' output as a starting point of a refinement proof to a more abstract version of the program. In the LEDA graph proofs, full functional correctness is not the goal, but simply showing that the program satisfies certain properties;

Table 8.2: Comparison of the specification sizes generated by Norrish's C-to-Isabelle parser [85] and AutoCorres of 5 large C programs.

Program	LoC	Functions	Lines of Spec		Average Term Size	
			Norrish	AutoCorres	Norrish	AutoCorres
seL4 kernel	10 121	551	20 576	11 928	318	112
CapDL SysInit	2 079	163	3 353	2 183	184	72
Piccolo kernel	936	56	1 748	1 198	372	182
eChronos	563	40	715	537	180	108
Schorr-Waite	19	1	120	57	766	311

- The Piccolo kernel is using the output of AutoCorres directly in order to prove properties about certain functions in the kernel; and finally,
- The seL4 project acts as a benchmark, both in terms of the scalability of AutoCorres and ensuring that AutoCorres supports a sufficient breadth of the C standard so that it can be used by real projects.

While there is clear interest in using the tool from independent verification projects, most of the projects above are still in their early stages; this is unsurprising, given the relatively young age of the AutoCorres tool. Unfortunately, more conclusive results about the effectiveness of AutoCorres on reducing effort in large-scale projects will only be available when these longer-running projects are completed.

In the meantime, we present some statistics about some of the above projects. Table 8.2 shows information from the projects we have access to, including the number of functions in the project and the lines of C code. We additionally provide the metrics *lines of specification* and *term size* for both the output of Norrish's C-to-Isabelle parser and AutoCorres. Since both tools directly emit terms in Isabelle/HOL's internal representation, we estimate the former number by using Isabelle/HOL's pretty printer for the generated definitions. The *term size* metric measures the number of nodes in the abstract syntax tree of a specification. While neither measurement is a perfect metric for specification complexity, the numbers reinforce our intuition that the output of AutoCorres is significantly simpler than that of the C parser with lines of specification ranging from 25% to 53% smaller and the term sizes ranging from 40% to 61% smaller.

Table 8.3 presents the time taken by Norrish's C-to-Isabelle parser and AutoCorres to carry out their respective translations. While AutoCorres has a longer running time than the C parser, for both tools this cost tends to be a one-off, where the results of the translation are saved and reused. Further, AutoCorres translates functions in parallel, so real time is significantly less than CPU time. We discuss in Section 9.7.1 possible approaches that could be used to further reduce the running time.

Table 8.3: Comparison of the time required by Norrish's C-to-Isabelle parser [85] and AutoCorres of 5 large C programs. Measurements are recorded on a 16-core 3.3GHz Intel Xeon E5-2643, with 128GiB of RAM. Measurements are the average of 5 runs. Memory usage values are peak memory usage across the run of both the C-to-Isabelle parser and AutoCorres; $1\,\mathrm{G} = 2^{30}$ bytes.

Program	LoC	Functions	CPU Time		Mem Usage
			Norrish	AutoCorres	
seL4 kernel	10 121	551	39.2 min	70.1 min	5.59 G
CapDL SysInit	2 079	163	3.0 min	23.3 min	2.82 G
Piccolo kernel	936	56	48.1 s	388.2 s	2.35 G
eChronos	563	40	15.0 s	67.8 s	2.24 G
Schorr-Waite	19	1	3.8 s	15.1 s	2.23 G

8.3 Conclusion

In this chapter, we have demonstrated in two case studies how the verification of low-level C code can now proceed at the same level of abstraction as previous verifications of idealised algorithms; further, the resulting proof scripts were significantly smaller than existing work that carried out similar proofs on the same C program.

We have also given an overview of how AutoCorres has been used and continues to be actively used in larger proof projects. Only once these project reach a more mature stage will we be able to gain a better understanding of the benefits and challenges provided by AutoCorres, but we are encouraged by the initial results.

Chapter Summary

- The goal of the tool AutoCorres is to both generate *useful* abstractions, and to be able to *scale* to real-world program sizes and features.
- To show that AutoCorres generates *useful* abstractions, we ported an existing proof of the Schorr-Waite algorithm [70] to AutoCorres. The original proof reasoned about the algorithm implemented in a very abstract language; our port to AutoCorres allowed us to prove the same property about a concrete C program, with minimal changes to reasoning.
- To show that AutoCorres is able to *scale* to real-world programs, we showed timing and output statistics of AutoCorres on the 10,000 line seL4 microkernel.
- AutoCorres is additionally being actively used in several other C projects, including a real-time operating system, a graph library, a system initialiser, and a memory allocator. This suggests that AutoCorres is flexible enough to support many real-world use-cases.

9 Conclusion

In this thesis, we have presented a technique called *specification abstraction*, where a conservative, low-level representation of a program is *automatically* and *verifiably* abstracted into a higher-level representation. These automated abstractions reduce cognitive complexity for verification engineers, and hence provide a boost in human productivity. The simpler representations also assist with mechanical reasoning, allowing automated reasoning tools to operate on a semantically distilled representation of the program.

An alternative approach commonly used to simplify program verification is to build more sophisticated reasoning tools—perhaps building a better VCG with more automation, for instance. In contrast, our specification abstraction approach is agnostic about the kind of verification to be conducted. It is easy to use a separation logic on top of our output, connect it to a VCG, use it as an intermediate step in a larger refinement proof, or use it as a base-level model for proving more complex properties, such as non-interference in the style of Murray et al [74]. Such flexibility is not just a theoretical nicety, but is being actively utilised, as demonstrated by the variety of ongoing projects using our work described in Section 8.2.

Finally, the algorithms described in this thesis are not simply theoretical but have been implemented and evaluated in a new tool named *AutoCorres*. AutoCorres is available under an open-source BSD-style license [48], and is actively being used on C verification projects both internal and external to the author's research group.

In the remainder of this chapter, we provide a summary of contributions made by this thesis, briefly describe how our work is applicable to other programming languages, and finally describe research challenges related to our work that remain open.

9.2 Thesis contributions

In this thesis we have developed formalisations and algorithms for automatically carrying out a variety of program abstractions. Importantly, we also automatically generate proofs that our translations are correct in Isabelle/HOL, providing the end-user with a formal connection between the low-level input program and our higher-level output specification.

In particular, in this thesis we have demonstrated the following verified transformations:

- Converting from a deeply embedded representation of an imperative program to a shallowly embedded monadic representation (Chapter 4);
- Lifting local variables from being modelled as part of the program's state to being represented in monadic bound variables (Chapter 5);
- Carrying out various optimisations to simplify program representations, including peephole optimisations (Section 4.5.1), exception elimination (Section 4.5.2), flow-sensitive optimisations (Section 5.2), and type strengthening (Section 5.3);
- Rewriting programs using word-based arithmetic, transforming them into programs that operate directly on unbounded integers and naturals (Chapter 6); and
- Rewriting programs that operate directly on a byte-level model of the heap, transforming them into programs operating on a Burstall-Bornat split-heap model (Chapter 7), without sacrificing the ability to carry out byte-level reasoning where necessary (Section 7.5).

We have implemented the algorithms described in this work in the tool AutoCorres, and have evaluated it using simple case studies (in each of the chapters of this thesis), and also by carrying out the following more significant evaluations:

- We have shown that pre-existing highly abstract proofs—such as a proof of an in-place linked list reversal function and a proof of the Schorr-Waite algorithm—can be ported to the output of AutoCorres with minimal effort (Section 8.1);
- We have demonstrated that AutoCorres is scalable to real-world project sizes by providing translation statistics on a number of larger projects (Section 8.2.1); and, finally,
- We have shown that AutoCorres' output is sufficiently flexible to be used in a variety of contexts, as demonstrated by the variety of ongoing projects where it is currently being used (Section 8.2).

While AutoCorres is still a relatively young tool—and hence has had little time to be used in larger completed projects—we are encouraged by initial evaluations and its uptake in ongoing projects.

9.3 Applicability to other languages

While our focus has been on the C programming language, we believe that many of the algorithms and rules in this thesis would be applicable for carrying out specification abstraction of other imperative languages, such as Java, C# or SPARK Ada.

In particular, the conversion from a deeply embedded representation to a shallowly embedded monadic representation would apply to every language, albeit with many of the implementation details differing. Similarly, local variable lifting, exception elimination, type strengthening and the optimisation passes would all be applicable to other imperative languages, though additional language-specific rules would be required for these phases to be fully effective.

Heap abstraction would thankfully no longer be required on higher-level languages such as Java and C#. The design and type-systems of these languages already imply that changes to an object of one type cannot affect another, nor can objects partially overlap.

The problem of finite word arithmetic, however, would still remain. Java and C# do not share C's concept of undefined behaviour when it comes to word arithmetic, instead defining signed overflow and underflow as the standard two's-complement result.[1] As discussed in Chapter 6, reasoning about arithmetic operations that may overflow is far more complex than reasoning about their unbounded counterparts. If users (*i*) do not rely on signed overflow, and (*ii*) are willing to prove this, then we could carry out word abstraction to translate finite word-based operations into unbounded integer operations, in much the same way that our work optionally abstracts unsigned arithmetic operations in C.

What is perhaps most striking is that the output of AutoCorres in our larger case studies in Section 8.1 is relatively programming language-agnostic—a user unfamiliar with AutoCorres would struggle to guess the underlying language the specifications were generated from. This suggests that, with sufficient tool support, reasoning about programs written in C should not be significantly harder than reasoning about equivalent programs written in higher-level languages such as Java or C#.

9.4 Trusting the C-to-Isabelle Parser

Our work uses the output of Norrish's C-to-Isabelle parser as its starting point. Our reasons for basing our own work on Norrish's C-to-Isabelle parser were pragmatic: Norrish's translation tool was mature, well tested, and targeted Isabelle/HOL directly.

Despite the tool's maturity and test framework, Norrish's C-to-Isabelle parser still consists of around 12 000 lines of unverified ML code and 13 500 lines of Isabelle code. Much of this code needs to be trusted. If the parser mistranslates a C construct, then any proof about the behaviour of the resulting Simpl is invalid. Similarly,

[1] That is, `INT_MAX + 1 = INT_MIN` and `-INT_MIN = INT_MIN`.

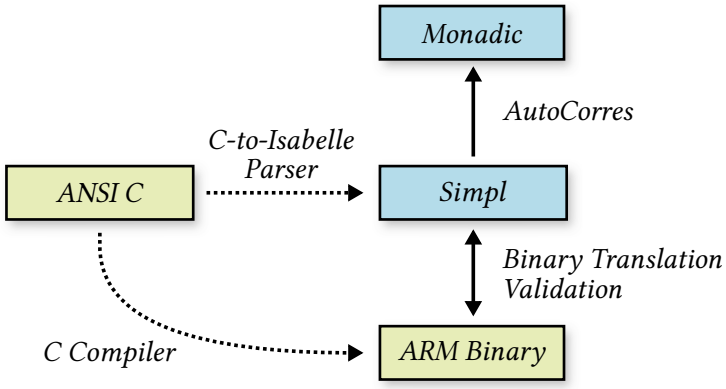


Figure 9.1: The connections between Norrish's C-to-Isabelle parser, Sewell et al.'s translation validation [95] and AutoCorres. Solid lines indicate formal connections between artefacts, while dashed lines indicate informal translations. Blue boxes indicate artefacts represented in Isabelle/HOL.

while AutoCorres produces a proof that its own abstraction is correct, this proof still fundamentally needs to assume that the underlying Simpl is correct.

In general, if we want to formally reason about C, there is no way to avoid placing at least *some* level of trust in whatever method we use to import C into our reasoning tool: at very least we must trust that our tool's formal interpretation of the informal C standard is correct. The goal then is to *reduce* the amount of trust that needs to be placed in such tools, by eliminating cleverness and attempting to implement as literal an implementation of the C standard as possible. Norrish's C-to-Isabelle parser was designed with this goal in mind: providing a literal translation of the C standard for the subset of C supported.

An alternative method of reducing the trust that needs to be placed in the C semantics is to establish a formal connection to the compiled binary code. One method is to develop a verified C compiler, such as the verified CompCert compiler [67]. Such a compiler can provide guarantees that the formal semantics of the C code match those of the generated binary, and hence proofs about the C code will also apply to the binary. Curiously, even using a verified compiler, verification engineers still have no guarantee that they are reasoning about the correct semantics of their C program—simply that any misinterpretation made by the compiler authors will be similarly reflected in the final binary.[2] In practice, such differences won't tend to matter if the verified compiler's toolchain is used end-to-end.

While Norrish's C-to-Isabelle parser isn't associated with a verified compiler, Sewell, Myreen and Klein [95] have shown that it is possible to automatically prove correspondence between the output of Norrish's parser and the binary output of the unverified GNU C Compiler (GCC). The process works as follows: (*i*) a C program is translated

[2] This is not an entirely theoretical issue: the CompCert semantics defines the result of signed overflow [27], while the C standard leaves it undefined. This means it is quite possible to prove a program correct using the CompCert semantics, only to have it crash when compiled with GCC—even under the assumption that GCC is entirely bug free.

into Simpl using Norrish's C-to-Isabelle parser; (*ii*) the Simpl output is then translated into a *graph language* which represents programs as a control flow graph, similar to that used internally by a compiler; (*iii*) the compiled code generated by GCC is also translated into the graph language; and (*iv*) finally, the two graphs (i.e., the Simpl and the output of GCC) are automatically proven to be equivalent using SMT solvers. If the proof of equivalence succeeds, users can be confident that the semantics generated by Norrish's C-to-Isabelle parser match the semantics of the code generated by GCC. In this case, users no longer need to trust the implementations of either GCC or Norrish's C-to-Isabelle parser; instead, trust is placed in the (much simpler) translation from assembly to the graph language. Again, there is still no guarantee that the verification engineer has the correct semantics of their C program, simply that Norrish's C-to-Isabelle parser and GCC agree.

When Sewell et al.'s work is connected to our own, we end up in the scenario depicted in Figure 9.1, where the output of AutoCorres is formally connected to the final binary. While Norrish's C-to-Isabelle parser does not support all of the C language, and Sewell et al.'s work still has limitations on the optimisation level of GCC;[3] when the entire toolchain *can* be successfully used, the user can have high confidence that theorems proven about their program in Isabelle/HOL apply to their concrete binary.

9.5 The Simpl language as an input

In the scenario described in the previous section and shown in Figure 9.1—i.e., the output of the C-to-Isabelle parser is used as an input to other tools, but never directly reasoned about by verification engineers—the Simpl language becomes northing more than a rendezvous point for the three tools: the C-to-Isabelle parser generates Simpl, while AutoCorres and Sewell et al.'s work consume it.

A natural question to ask would be: "is the Simpl generated by Norrish's C-to-Isabelle parser the best choice for such a rendezvous point?" Two areas where we feel Simpl presents unnecessary difficulties for both AutoCorres and Sewell et al.'s work are as follows:

Shallowly embedded expressions Simpl is a language with deeply embedded statements and shallowly embedded expressions. This latter design decision was made to ease reasoning about Simpl programs. As discussed in Chapter 4, reasoning about shallowly embeddings of programs tends to be easier for verification engineers than reasoning about deep embeddings. At the time that both Simpl and Norrish's C-to-Isabelle parser were developed, there was an expectation that the output of the parser would be directly reasoned about by verification engineers. Thus, using a shallow embedding for expressions was the pragmatic decision.

If we treat Simpl as nothing more than a rendezvous point for several automatic tools, the benefits of using shallow embeddings for expressions disappear, while many

[3] Sewell et al.'s work is ongoing at time of writing, attempting to address these limitations.

disadvantages remain. In particular, functions written in Isabelle cannot parse shallowly embedded terms; instead, we must write the different phases of AutoCorres in ML simply to analyse the terms. While verifying Isabelle functions (i.e., functions written directly in Isabelle's logic) is relatively easy, verifying ML implementations is much harder. This makes it difficult provide any guarantees about the correctness or completeness of functions in AutoCorres. In our work, we resort to generating certificates showing that each individual output generated by AutoCorres is correct after the fact.

In contrast, if expressions were deeply embedded, a standard Isabelle function could be written to analyse and translate it into other forms. Proofs about this function could be carried out to show both its correctness and completeness once and for all.

A C-specific representation While the Simpl language attempts to provide all the features required by an imperative languages such as C, in practice there remain mismatches between the features provided by Simpl and those required by C. One such example are the behaviour of C function calls (including stacking of local variables and extraction of return values), which have a non-trivial translation from C to Simpl, as described in Section 4.4.

In a world where verification engineers no longer need to reason directly about the output of Norrish's C-to-Isabelle parser, we could simply have the parser generate a deeply embedded C-specific abstract syntax tree (AST) of the input C program. Similar parsers exist, as described in Section 2.1.3 and Section 9.7.4, albeit not for Isabelle/HOL. By having a C-specific representation of the language, the mismatch between the input C language and the output logical language could be eliminated. While such a representation would be more difficult to manually work with, automated tools such as AutoCorres could alleviate the problem by presenting the end-user with a simpler representation.

9.6 Output stability

Our experience in verifying large software systems suggests that the process of program verification is not simply a matter of (*i*) writing a piece of software; and then (*ii*) proving it correct. Rather, it is an iterative process, where software is continuously tweaked and modified during the verification process in response to feedback from verification engineers. Perhaps the most obvious example of this is when software must be modified to fix bugs revealed during verification; but software may also be modified mid-verification simply to ease verification, such as by simplifying some aspect of the code or tightening a program invariant. Even once verification of a program has completed, software may continue to change when new features are added, or existing features are modified.

In the context of changing input programs, one aspect that is important for tools like AutoCorres is *output stability*. That is, will small changes to the input program result in small changes to the output specification? For completely automated verification tools that simply state 'correct' or 'incorrect', output stability is not a particularly pressing concern; but for tools like AutoCorres that generate an artefact that will be

manually reasoned about, it would be unacceptable if each time a small change was made to the input program, the entire proof on AutoCorres' output needed to be restarted.

While we don't have firm numbers about the stability of AutoCorres' output, our experience suggests that—with the exception of the type strengthening process, discussed below—the output of AutoCorres tends to be stable, with small changes to input programs typically resulting in small changes to output programs. As most of the translation phases carried out by AutoCorres only depend on information within each individual function, if a single function is modified—while that individual function may be translated differently by AutoCorres—other functions will be unaffected.

The primary exception to this is the type strengthening phase of AutoCorres, described in Section 5.3. In particular, type strengthening attempts to determine the strongest type a function may be represented as. For example, if a function does not write to the heap, AutoCorres will represent that function using the stronger option monad type, instead of the weaker state monad type. One limitation of type strengthening is that a function cannot be represented in a type stronger than that of the functions it calls; that is, if a function f calls a second function g, then f's type cannot be stronger than g's.

A problem arises if a function is modified by the user in such a way that it can no longer be represented in its current type. For example, a function f previously represented by AutoCorres using the option monad might be modified in such a way that it now writes to the heap, so that AutoCorres can only represent it using the state monad. Not only will the type of f change—requiring changes in all proofs about f—but also the type of any functions *calling* f, any functions calling *those* functions, and so on.

One method we have implemented to somewhat mitigate this problem is to introduce *type pinning*, where the user may request that certain functions always be translated to a particular type. While a function cannot be pinned to a type stronger than what is necessary to represent it—that is, a function that modifies the heap fundamentally cannot be represented using the option monad, for example—when developers can foresee that a function will eventually be modified and require a weaker type than it currently does, type pinning can reduce the maintenance burden later down the line.

9.7 Future work

In this section, we present potential areas of improvement and research that stem from the work presented in this thesis.

9.7.1 Improving performance

AutoCorres is designed to scale to large projects. In its current state, it is able to process 10,000-line C projects in a few hours of CPU time. This cost tends to be a one-off: a verification engineer runs AutoCorres on an input C file once, and then

carries out verification on the result. None the less, it would be ideal if AutoCorres could carry out these translations faster; this would allow AutoCorres to scale to larger projects, and also improve engineer productivity when changes to C code are required.

Perhaps the greatest bottleneck in the existing code is the flow-sensitive optimisation phase, which we estimate requires around two-thirds of the translation time. We discuss the reasons for the lacklustre performance and provide alternative solutions below. The remaining bottlenecks arise primarily from our requirement that we validate all of our translations using the Isabelle/HOL proof kernel.

One possible approach to speed up our implementation is to use *proof by reflection*. In this approach, we (*i*) prove that the implementation of AutoCorres is correct in a theorem prover; (*ii*) translate the implementation to an executable format; and then (*iii*) use this proven version of AutoCorres to carry out our abstraction algorithms, relying on the correctness of our implementation instead of generating an explicit LCF-style proof of each program we translate. This process needn't be done for the entire AutoCorres implementation, but can be selectively used for particularly problematic phases in AutoCorres.

One theoretical objection to this approach is that the strict LCF-style proof that is currently generated by AutoCorres—which connects the input C to the output specification—is no longer available. An increased level of trust must be placed in the code that compiles AutoCorres, and there are limited opportunities for external proof-replay.

An alternative approach to improving performance that retains a strict LCF-style proof is to create two versions of each procedure in AutoCorres: an informal *quick and dirty* version that quickly calculates the results of each AutoCorres phase without attempting to generate proofs, and a *strict* mode that generates a full proof. The quick and dirty phase could carry out the specification abstraction process quickly, spawning new threads of execution responsible for double-checking the results using strict proof checking. These new threads could run in the background and could also be arbitrarily parallelised.[4]

Maintaining two versions of each procedure in AutoCorres and ensuring they remain synchronised would be a huge maintenance burden. A potential solution is to *automatically generate* the quick and dirty implementation. For the phases of AutoCorres that use sets of rules to carry out calculations (such as word abstraction, heap abstraction, type strengthening, and so on), the informal procedures could be generated automatically from the rules. For the phases of AutoCorres that rely on judgements from Isabelle/HOL's simplifier to progress (such as the peephole optimisations), again the set of rules used by the simplifier could be compiled into a procedure to informally determine the judgements ahead of time. For the remainder of the AutoCorres phases, proofs are already carried out asynchronously, or could be adjusted to do so with relatively little effort.

[4] In particular, we desire an end-to-end proof to be checked by n threads. The quick and dirty thread generates $n - 1$ 'stepping stones'. Then, the ith thread checks that the proof from the ith to the $(i + 1)$th is correct. Once all n threads have finished, the proofs are joined together to form a single end-to-end proof.

9.7.2 Implementing abstract interpretation

The current flow-sensitive optimisations described in Section 5.2, while *relatively* effective, still leave much to be desired. There are two main limitations in the current implementation.

The first limitation is that facts about the current function are gathered from guard, condition, and whileLoop statements, which are then unceremoniously passed into Isabelle/HOL's simplifier where they can ideally be used to simplify later statements. In the current implementation, the number of facts given to the simplifier grows linearly with the function size,[5] significantly slowing down the flow-sensitive optimisation phase of AutoCorres, particularly in larger functions.

The second problem is that the current implementation of flow-sensitive optimisation frequently 'forgets' facts it learnt earlier in the function. For example, after each condition block, AutoCorres throws away learnt information to avoid a potential exponential-blowup in the number of facts it tracks. Additionally, AutoCorres is incapable of using facts learnt in one iteration of a whileLoop's body in the next iteration.

Abstract interpretation [34] is a well-known solution to both of these problems, where variables in a function are interpreted in an abstract domain. For instance, we may decide to keep track of the maximum and minimum possible values of each variable, but nothing more. By carefully choosing an appropriate abstract domain, sound analysis of functions can be carried out without an explosion in the number of facts being stored. Additionally, the *widening* and *narrowing* procedures of abstract interpretation allow loops to be better analysed, with the possible output values of the loop soundly calculated. Cachera and Pichardie [25] have shown that it is possible to carry out abstract interpretation from within an interactive theorem prover for a simple language. Extending this to support the complexities of C would be a practical exercise towards improving the quality of our specification abstraction. For example, abstract interpretation could be used to carry out a range analysis of word-based variables to automatically eliminate guards generated from the use of word arithmetic which can be shown to always be true.

9.7.3 Data structure abstractions

Currently AutoCorres is capable of soundly abstracting the system heap from using a byte-level model to a Burstall-Bornat style model. The next question is, could we go further and abstract entire data structures, such as lists or trees?

One tractable approach to this problem would be to craft a set of APIs in C for data structure manipulation. For instance, we could introduce an API function `list_concat` to add an item to a list, and an API macro `LIST_FORALL` to enumerate a list. Large C projects such as the Linux kernel already use such an internal API as a matter of good engineering practice.

[5] For a function of length n, $\mathcal{O}(n)$ facts will be generated, each of which will be used $\mathcal{O}(n)$ times by the simplifier. This results in $\mathcal{O}(n^2)$ units of work being given to the simplifier, which itself has a non-trivial time complexity.

Next, we detect these data structures on the heap as well as the calls to their associated API functions, abstracting them into their logical equivalents. For instance, a call to `list_append(a, b)` would be translated into the logically equivalent Isabelle/HOL list append operation *a* @ *b*.

For programs that heavily manipulate data structures in memory, having a small library of data structures that can be automatically abstracted into their logical equivalent operations would significantly ease reasoning.

9.7.4 An extended C subset

Our work uses Norrish's C-to-Isabelle parser [84, 85] as its starting point. Norrish's parser has the advantage that it supports a sufficiently large subset of C to allow real-world projects to be verified, such as the seL4 microkernel [57, 81, 82]. Also, as previously discussed in Section 9.5, Sewell, Myreen and Klein [95] have also proven semantic equivalence between the output of Norrish's C parser and the binary output of the GNU C Compiler [99]. This latter work reduces the amount of trust that must be placed in both Norrish's C parser and the GNU C Compiler.

Other C semantics have also been developed that are both trustworthy and cover a greater subset of the C standard than Norrish's semantics. The first is the verified CompCert C compiler [67], which has a formal proof in the Coq interactive theorem prover that the C semantics match the semantics of the binary code generated by the compiler. The second is the extensive C semantics developed by Ellison and Roşu [42], which, although not formally connected to a C compiler, have been extensively tested and shown to empirically match the GNU C Compiler.

While it would require significant engineering effort, it would be possible to modify AutoCorres to work on Isabelle/HOL translations of these two models of C. An alternative approach would be to reimplement the algorithms and logical frameworks described in this work to use the CompCert C semantics and operate in the logic of the Coq interactive theorem prover. If completed, the end result would be a formal link between the output of a AutoCorres-like tool and the binary output generated by the CompCert compiler.

Porting our algorithms would not be without its challenges. The early phases of AutoCorres—such as converting from the deeply embedded Simpl language to a shallowly embedded monadic representation—are highly specific to Norrish's C-to-Isabelle parser. Similarly, the heap abstraction phase of AutoCorres would need to take into account the different models of a byte-level heap used by these projects. For many of the phases of AutoCorres, however, our existing work could be used relatively unchanged.

9.8 Final words

In this thesis, we have described a technique called *specification abstraction*, which allows high-level reasoning to take place on conservative, low-level models of programming languages without sacrificing soundness. It is our hope that future formal

verification tools increasingly use verified specification abstraction—either explicitly exposed to the user, or simply as an internal transformation—to ensure that they are able to provide strong guarantees about software correctness.

A Appendices

A.1 Big-step semantics of Simpl

The following rules describe the big-step semantics of Schirmer's *Simpl* language [92, 93], which is generated by Norrish's C-to-Isabelle parser and is the starting point of our work.

Basic control flow

$$\frac{}{\Gamma \vdash \langle \text{Skip}, \text{Normal } s\rangle \Rightarrow \text{Normal } s} \qquad \frac{}{\Gamma \vdash \langle \text{Basic } f, \text{Normal } s\rangle \Rightarrow \text{Normal } (f\ s)}$$

$$\frac{\Gamma \vdash \langle c_1, \text{Normal } s\rangle \Rightarrow s' \quad \Gamma \vdash \langle c_2, s'\rangle \Rightarrow t}{\Gamma \vdash \langle c_1;; c_2, \text{Normal } s\rangle \Rightarrow t}$$

$$\frac{s \in b \quad \Gamma \vdash \langle c_1, \text{Normal } s\rangle \Rightarrow t}{\Gamma \vdash \langle \text{Cond } b\ c_1\ c_2, \text{Normal } s\rangle \Rightarrow t} \qquad \frac{s \notin b \quad \Gamma \vdash \langle c_2, \text{Normal } s\rangle \Rightarrow t}{\Gamma \vdash \langle \text{Cond } b\ c_1\ c_2, \text{Normal } s\rangle \Rightarrow t}$$

$$\frac{\begin{array}{c} s \in b \\ \Gamma \vdash \langle c, \text{Normal } s\rangle \Rightarrow s' \\ \Gamma \vdash \langle \text{While } b\ c, s'\rangle \Rightarrow t \end{array}}{\Gamma \vdash \langle \text{While } b\ c, \text{Normal } s\rangle \Rightarrow t} \qquad \frac{s \notin b}{\Gamma \vdash \langle \text{While } b\ c, \text{Normal } s\rangle \Rightarrow \text{Normal } s}$$

$$\frac{\begin{array}{c} \Gamma\ p = \text{Some } body \\ \Gamma \vdash \langle body, \text{Normal } s\rangle \Rightarrow t \end{array}}{\Gamma \vdash \langle \text{Call } p, \text{Normal } s\rangle \Rightarrow t} \qquad \frac{\Gamma\ p = \text{None}}{\Gamma \vdash \langle \text{Call } p, \text{Normal } body\rangle \Rightarrow \text{Stuck}}$$

Exceptional control flow

$$\frac{}{\Gamma \vdash \langle \mathsf{Throw},\ \mathsf{Normal}\ s\rangle \Rightarrow \mathsf{Abrupt}\ s} \qquad \frac{}{\Gamma \vdash \langle c,\ \mathsf{Abrupt}\ s\rangle \Rightarrow \mathsf{Abrupt}\ s}$$

$$\frac{\Gamma \vdash \langle c_1,\ \mathsf{Normal}\ s\rangle \Rightarrow t \qquad \nexists s.\ t = \mathsf{Abrupt}\ s}{\Gamma \vdash \langle \mathsf{Catch}\ c_1\ c_2,\ \mathsf{Normal}\ s\rangle \Rightarrow t} \qquad \frac{\Gamma \vdash \langle c_1,\ \mathsf{Normal}\ s\rangle \Rightarrow \mathsf{Abrupt}\ s' \qquad \Gamma \vdash \langle c_2,\ \mathsf{Normal}\ s'\rangle \Rightarrow t}{\Gamma \vdash \langle \mathsf{Catch}\ c_1\ c_2,\ \mathsf{Normal}\ s\rangle \Rightarrow t}$$

Guards

$$\frac{s \in g \qquad \Gamma \vdash \langle c,\ \mathsf{Normal}\ s\rangle \Rightarrow t}{\Gamma \vdash \langle \mathsf{Guard}\ f\ g\ c,\ \mathsf{Normal}\ s\rangle \Rightarrow t} \qquad \frac{s \notin g}{\Gamma \vdash \langle \mathsf{Guard}\ f\ g\ c,\ \mathsf{Normal}\ s\rangle \Rightarrow \mathsf{Fault}\ f}$$

$$\frac{}{\Gamma \vdash \langle c,\ \mathsf{Fault}\ f\rangle \Rightarrow \mathsf{Fault}\ f}$$

Non-determinism and dynamic execution

$$\frac{(s,\ t) \in r}{\Gamma \vdash \langle \mathsf{Spec}\ r,\ \mathsf{Normal}\ s\rangle \Rightarrow \mathsf{Normal}\ t} \qquad \frac{\forall t.\ (s,\ t) \notin r}{\Gamma \vdash \langle \mathsf{Spec}\ r,\ \mathsf{Normal}\ s\rangle \Rightarrow \mathsf{Stuck}}$$

$$\frac{}{\Gamma \vdash \langle c,\ \mathsf{Stuck}\rangle \Rightarrow \mathsf{Stuck}}$$

$$\frac{\Gamma \vdash \langle c\ s,\ \mathsf{Normal}\ s\rangle \Rightarrow t}{\Gamma \vdash \langle \mathsf{DynCom}\ c,\ \mathsf{Normal}\ s\rangle \Rightarrow t}$$

A.2 Termination of Simpl programs

The following rules describe the termination semantics of Schirmer's *Simpl* language [92, 93].

Basic control flow

$$\frac{}{\Gamma \vdash \text{Skip} \downarrow \text{Normal } s} \qquad \frac{}{\Gamma \vdash \text{Basic } f \downarrow \text{Normal } s}$$

$$\frac{\Gamma \vdash c_1 \downarrow \text{Normal } s \quad \forall s'.\ \Gamma \vdash \langle c_1,\ \text{Normal } s\rangle \Rightarrow s' \longrightarrow \Gamma \vdash c_2 \downarrow s'}{\Gamma \vdash c_1;;\ c_2 \downarrow \text{Normal } s}$$

$$\frac{s \in b \quad \Gamma \vdash c_1 \downarrow \text{Normal } s}{\Gamma \vdash \text{Cond } b\ c_1\ c_2 \downarrow \text{Normal } s} \qquad \frac{s \notin b \quad \Gamma \vdash c_2 \downarrow \text{Normal } s}{\Gamma \vdash \text{Cond } b\ c_1\ c_2 \downarrow \text{Normal } s}$$

$$\frac{s \in b \qquad \Gamma \vdash c \downarrow \text{Normal } s \quad \forall s'.\ \Gamma \vdash \langle c,\ \text{Normal } s\rangle \Rightarrow s' \longrightarrow \Gamma \vdash \text{While } b\ c \downarrow s'}{\Gamma \vdash \text{While } b\ c \downarrow \text{Normal } s}$$

$$\frac{s \notin b}{\Gamma \vdash \text{While } b\ c \downarrow \text{Normal } s} \qquad \frac{\Gamma\ p = \text{Some } body \quad \Gamma \vdash body \downarrow \text{Normal } s}{\Gamma \vdash \text{Call } p \downarrow \text{Normal } s} \qquad \frac{\Gamma\ p = \text{None}}{\Gamma \vdash \text{Call } p \downarrow \text{Normal } s}$$

Exceptional control flow

$$\frac{}{\Gamma \vdash \text{Throw} \downarrow \text{Normal } s} \qquad \frac{}{\Gamma \vdash c \downarrow \text{Abrupt } s}$$

$$\frac{\Gamma \vdash c_1 \downarrow \text{Normal } s \quad \forall s'.\ \Gamma \vdash \langle c_1,\ \text{Normal } s\rangle \Rightarrow \text{Abrupt } s' \longrightarrow \Gamma \vdash c_2 \downarrow \text{Normal } s'}{\Gamma \vdash \text{Catch } c_1\ c_2 \downarrow \text{Normal } s}$$

Guards

$$\frac{s \in g \quad \Gamma \vdash c \downarrow \text{Normal } s}{\Gamma \vdash \text{Guard } f\ g\ c \downarrow \text{Normal } s} \qquad \frac{s \notin g}{\Gamma \vdash \text{Guard } f\ g\ c \downarrow \text{Normal } s} \qquad \frac{}{\Gamma \vdash c \downarrow \text{Fault } f}$$

Non-determinism and dynamic execution

$$\frac{}{\Gamma \vdash \text{Spec } r \downarrow \text{Normal } s} \qquad \frac{}{\Gamma \vdash c \downarrow \text{Stuck}} \qquad \frac{\Gamma \vdash c\ s \downarrow \text{Normal } s}{\Gamma \vdash \text{DynCom } c \downarrow \text{Normal } s}$$

Bibliography

[1] E. Alkassar, S. Böhme, K. Mehlhorn and C. Rizkallah. 'A Framework for the Verification of Certifying Computations'. In: *Journal of Automated Reasoning* 52.3 (2014), pages 241–273. DOI: 10.1007/s10817-013-9289-2.

[2] E. Alkassar, M. Hillebrand, D. Leinenbach, N. Schirmer, A. Starostin and A. Tsyban. 'Balancing the Load — Leveraging a Semantics Stack for Systems Verification'. In: *Journal of Automated Reasoning: Special Issue on Operating System Verification* 42, Numbers 2–4 (2009), pages 389–454. DOI: 10.1007/s10817-009-9123-z.

[3] J. Andronick, B. Chetali and C. Paulin-Mohring. 'Formal Verification of Security Properties of Smart Card Embedded Source Code'. In: *Proceedings of the International Symposium on Formal Methods (FM)*. Volume 3582. LNCS. 2005, pages 302–317. DOI: 10.1007/11526841_21.

[4] A. W. Appel. 'VeriSmall: Verified Smallfoot Shape Analysis'. In: *Proceedings of the 1st International Conference on Certified Programs and Proofs*. Volume 7086. LNCS. 2011, pages 231–246. DOI: 10.1007/978-3-642-25379-9_18.

[5] A. W. Appel. *Verification of a Cryptographic Primitive: SHA-256*. Accessed July 2014. 2014. URL: http://www.cs.princeton.edu/~appel/papers/verif-sha.pdf.

[6] T. Ball, E. Bounimova, R. Kumar and V. Levin. 'SLAM2: Static driver verification with under 4% false alarms'. In: *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*. 2010, pages 35–42. ISBN: 978-1-4577-0734-6.

[7] T. Ball, A. Podelski and S. K. Rajamani. 'Relative completeness of abstraction refinement for software model checking'. In: *Tools and Algorithms for the Construction and Analysis of Systems*. 2002, pages 158–172. DOI: 10.1007/3-540-46002-0_12.

[8] C. Ballarin. 'Locales and Locale Expressions in Isabelle/Isar'. In: *Types for Proofs and Programs*. Volume 3085. LNCS. 2004, pages 34–50. DOI: 10.1007/978-3-540-24849-1_3.

[9] C. Barrett and C. Tinelli. 'CVC3'. In: *Computer Aided Verification*. Volume 4590. LNCS. 2007, pages 298–302. DOI: 10.1007/978-3-540-73368-3_34.

[10] J. Berdine, C. Calcagno and P. W. O'Hearn. 'Smallfoot: Modular automatic assertion checking with separation logic'. In: *Proceedings of the 4th International Conference on Formal Methods for Components and Objects*. 2005, pages 115–137. DOI: 10.1007/11804192_6.

[11] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. DOI: 10.1007/978-3-662-07964-5.

[12] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux and X. Rival. 'A static analyzer for large safety-critical software'. In: *SIGPLAN Notices*. Volume 38. 5. ACM. 2003, pages 196–207. DOI: 10.1145/781131.781153.

[13] S. Blazy, Z. Dargaye and X. Leroy. 'Formal Verification of a C Compiler Front-End'. In: *Proceedings of the 14th International Symposium on Formal Methods (FM)*. Volume 4085. LNCS. 2006, pages 460–475. DOI: 10.1007/11813040_31.

[14] S. Blazy, V. Laporte, A. Maroneze and D. Pichardie. 'Formal Verification of a C Value Analysis Based on Abstract Interpretation'. In: *Static Analysis*. Volume 7935. LNCS. 2013, pages 324–344. DOI: 10.1007/978-3-642-38856-9_18.

[15] S. Blazy and X. Leroy. 'Mechanized Semantics for the C-light Subset of the C Language'. In: *Journal of Automated Reasoning* 43.3 (2009), pages 263–288. DOI: 10.1007/s10817-009-9148-3.

[16] S. Böhme, K. R. M. Leino and B. Wolff. 'HOL-Boogie — An Interactive Prover for the Boogie Program-Verifier'. In: *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*. Volume 5170. LNCS. 2008, pages 150–166. DOI: 10.1007/978-3-540-71067-7_15.

[17] S. Böhme, M. Moskal, W. Schulte and B. Wolff. 'HOL-Boogie — An Interactive Prover-Backend for the Verifying C Compiler'. In: *Journal of Automated Reasoning* 44.1–2 (2010), pages 111–144. DOI: 10.1007/s10817-009-9142-9.

[18] R. Bornat. 'Proving pointer programs in Hoare Logic'. In: *Proceedings of the 5th Mathematics of Program Construction*. Volume 1837. LNCS. 2000, pages 102–126. DOI: 10.1007/10722010_8.

[19] R. J. Boulton, A. Gordon, M. J. C. Gordon, J. Harrison, J. Herbert and J. V. Tassel. 'Experience with Embedding Hardware Description Languages in HOL'. In: *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*. 1992, pages 129–156. ISBN: 0-444-89686-4.

[20] A. Boyton, J. Andronick, C. Bannister, M. Fernandez, X. Gao, D. Greenaway, G. Klein, C. Lewis and T. Sewell. 'Formally Verified System Initialisation'. In: *Proceedings of the 15th International Conference on Formal Engineering Methods*. 2013, pages 70–85. DOI: 10.1007/978-3-642-41202-8_6.

[21] L. Bulwahn, A. Krauss, F. Haftmann, L. Erkök and J. Matthews. 'Imperative Functional Programming with Isabelle/HOL'. In: *Theorem Proving in Higher Order Logics*. Volume 5170. LNCS. 2008, pages 134–149. DOI: 10.1007/978-3-540-71067-7_14.

[22] L. Bulwahn, A. Krauss and T. Nipkow. 'Finding Lexicographic Orders for Termination Proofs in Isabelle/HOL'. In: *Theorem Proving in Higher Order Logics*. Volume 4732. LNCS. 2007, pages 38–53. DOI: 10.1007/978-3-540-74591-4_5.

[23] R. Burstall. 'Some techniques for proving correctness of programs which alter data structures'. In: *Machine Intelligence 7*. 1972, pages 23–50.

[24] S. Böhme, A. C. J. Fox, T. Sewell and T. Weber. 'Reconstruction of Z3's Bit-Vector Proofs in HOL4 and Isabelle/HOL'. In: *Certified Programs and Proofs*. Volume 7086. LNCS. 2011, pages 183–198. DOI: 10.1007/978-3-642-25379-9_15.

[25] D. Cachera and D. Pichardie. 'A Certified Denotational Abstract Interpreter'. English. In: *Interactive Theorem Proving*. Volume 6172. LNCS. 2010, pages 9–24. DOI: 10.1007/978-3-642-14052-5_3.

[26] C. Calcagno, D. Distefano, P. W. O'Hearn and H. Yang. 'Compositional Shape Analysis by Means of Bi-Abduction'. In: *Journal of the ACM* 58.6 (2011), 26:1–26:66. DOI: 10.1145/2049697.2049700.

[27] B. Campbell. 'An Executable Semantics for CompCert C'. English. In: *International Conference on Certified Programs and Proofs*. Volume 7679. LNCS. 2012, pages 60–75. DOI: 10.1007/978-3-642-35308-6_8.

[28] S. Chaki, E. M. Clarke, A. Groce, S. Jha and H. Veith. 'Modular verification of software components in C'. In: *IEEE Transactions on Software Engineering* 30.6 (2004), pages 388–402. DOI: 10.1109/TSE.2004.22.

[29] E. Clarke, O. Grumberg, S. Jha, Y. Lu and H. Veith. 'Counterexample-guided abstraction refinement for symbolic model checking'. In: *Journal of the ACM* 50.5 (2003), pages 752–794. DOI: 10.1145/876638.876643.

[30] D. Cock, G. Klein and T. Sewell. 'Secure Microkernels, State Monads and Scalable Refinement'. In: *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*. 2008, pages 167–182. DOI: 10.1007/978-3-540-71067-7_16.

[31] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte and S. Tobies. 'VCC: A Practical System for Verifying Concurrent C'. In: *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*. Volume 5674. LNCS. 2009, pages 23–42. DOI: 10.1007/978-3-642-03359-9_2.

[32] E. Cohen, M. Moskal, S. Tobies and W. Schulte. 'A precise yet efficient memory model for C'. In: *Proceedings of the 4th Systems Software Verification*. Volume 254. Electronic Notes in Theoretical Computer Science. 2009, pages 85–103. DOI: 10.1016/j.entcs.2009.09.061.

[33] P. Cousot and R. Cousot. 'Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints'. In: *Proceedings of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 1977, pages 238–252. DOI: 10.1145/512950.512973.

[34] P. Cousot and R. Cousot. 'Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints'. In: *Proceedings of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 1977, pages 238–252. DOI: 10.1145/512950.512973.

[35] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux and X. Rival. 'The ASTRÉE analyzer'. In: *Programming Languages and Systems*. Volume 3444. LNCS. 2005, pages 21–30. DOI: 10.1007/978-3-540-31987-0_3.

[36] M. Daum, N. Billing and G. Klein. 'Concerned with the Unprivileged: User Programs in Kernel Refinement'. In: *Formal Aspects of Computing* (2014). To appear, pages 1–25. DOI: 10.1007/s00165-014-0296-9.

[37] M. Daum, S. Maus, N. Schirmer and M. N. Seghir. 'Integration of a software model checker into Isabelle'. In: *Proceedings of the 12th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*. 2005, pages 381–395. DOI: 10.1007/11591191_27.

[38] J. Dawson. 'Isabelle Theories for Machine Words'. In: *Electronic Notes in Theoretical Computer Science* 250.1 (2009). Proceedings of the Seventh International Workshop on Automated Verification of Critical Systems (AVoCS 2007), pages 55–70. DOI: 10.1016/j.entcs.2009.08.005.

[39] D. Detlefs, G. Nelson and J. B. Saxe. 'Simplify: A Theorem Prover for Program Checking'. In: *Journal of the ACM* 52.3 (2005), pages 365–473. DOI: 10.1145/1066100.1066102.

[40] D. Distefano, P. W. O'Hearn and H. Yang. 'A local shape analysis based on separation logic'. In: *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2006, pages 287–302. DOI: 10.1007/11691372_19.

[41] J. Dodds and A. W. Appel. 'Mostly Sound Type System Improves a Foundational Program Verifier'. In: *International Conference on Certified Programs and Proofs*. Volume 8307. LNCS. 2013, pages 17–32. DOI: 10.1007/978-3-319-03545-1_2.

[42] C. Ellison and G. Roşu. 'An Executable Formal Semantics of C with Applications'. In: *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2012, pages 533–544. DOI: 10.1145/2103621.2103719.

[43] J.-C. Filliâtre and C. Marché. 'Multi-prover Verification of C Programs'. In: *Formal Methods and Software Engineering*. Volume 3308. LNCS. 2004, pages 15–29. DOI: 10.1007/978-3-540-30482-1_10.

[44] J.-C. Filliâtre and A. Paskevich. 'Why3 — Where Programs Meet Provers'. English. In: *Programming Languages and Systems*. Volume 7792. LNCS. 2013, pages 125–128. DOI: 10.1007/978-3-642-37036-6_8. URL: http://dx.doi.org/10.1007/978-3-642-37036-6_8.

[45] H. Gast. 'Reasoning about Memory Layouts'. In: *Proceedings of the 16th International Symposium on Formal Methods (FM)*. Volume 5850. LNCS. 2009, pages 628–643. DOI: 10.1007/978-3-642-05089-3_40.

[46] M. J. C. Gordon, R. Milner and C. P. Wadsworth. *Edinburgh LCF*. Volume 78. LNCS. Springer, 1979. DOI: 10.1007/3-540-09724-4.

[47] M. Gordon, R. Milner, L. Morris, M. Newey and C. Wadsworth. 'A Metalanguage for Interactive Proof in LCF'. In: *Proceedings of the 5th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 1978, pages 119–130. DOI: 10.1145/512760.512773.

[48] D. Greenaway. *AutoCorres tool*. Accessed August 2014. 2014. DOI: 10.5281/zenodo.13342. URL: http://ssrg.nicta.com.au/projects/TS/autocorres/.

[49] D. Greenaway, J. Andronick and G. Klein. 'Bridging the Gap: Automatic Verified Abstraction of C'. In: *Proceedings of the 3rd International Conference on Interactive Theorem Proving*. Volume 7406. LNCS. 2012, pages 99–115. DOI: 10.1007/978-3-642-32347-8_8.

[50] D. Greenaway, J. Lim, J. Andronick and G. Klein. 'Don't Sweat the Small Stuff: Formal Verification of C Code Without the Pain'. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2014, pages 429–439. DOI: 10.1145/2594291.2594296.

[51] Y. Gurevich and J. K. Huggins. 'The Semantics of the C Programming Language'. In: *Computer Science Logic*. Volume 702. LNCS. 1993, pages 274–308. DOI: 10.1007/3-540-56992-8_17.

[52] T. A. Henzinger, R. Jhala, R. Majumdar and G. Sutre. 'Software Verification with Blast'. In: *Proceedings of the 10th SPIN Workshop on Model Checking Software*. Volume 2648. LNCS. 2003, pages 235–239. DOI: 10.1007/3-540-44829-2_17.

[53] P. Herms, C. Marché and B. Monate. 'A Certified Multi-prover Verification Condition Generator'. In: *Verified Software: Theories, Tools, Experiments*. Volume 7152. LNCS. 2012, pages 2–17. DOI: 10.1007/978-3-642-27705-4_2.

[54] T. Hubert and C. Marché. 'A case study of C source code verification: the Schorr-Waite algorithm'. In: *Proceedings of the 3rd IEEE International Conference on Software Engineering and Formal Methods*. 2005, pages 190–199. DOI: 10.1109/SEFM.2005.1.

[55] ISO. *ISO/IEC 9899:2011 Information technology — Programming languages — C*. "The C11 Standard". International Organization for Standardization, 2011.

[56] B. W. Kernighan and D. M. Ritchie. *The C programming language*. Volume 2. Prentice-Hall, 1988. ISBN: 0-13-110370-9.

[57] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch and S. Winwood. 'seL4: Formal Verification of an OS Kernel'. In: *ACM Symposium on Operating Systems Principles*. 2009, pages 207–220. DOI: 10.1145/1629575.1629596.

[58] G. Klein, R. Kolanski and A. Boyton. 'Mechanised Separation Algebra'. In: *Interactive Theorem Proving (ITP)*. 2012, pages 332–337. DOI: 10.1007/978-3-642-32347-8_22.

[59] G. Klein, R. Kolanski and A. Boyton. 'Separation Algebra'. In: *Archive of Formal Proofs* (2012). Formal proof development. ISSN: 2150-914x. URL: http://afp.sf.net/entries/Separation_Algebra.shtml.

[60] A. Krauss. 'Partial and Nested Recursive Function Definitions in Higher-order Logic'. In: *Journal of Automated Reasoning* 44.4 (2010), pages 303–336. DOI: 10.1007/s10817-009-9157-2.

[61] R. Krebbers, X. Leroy and F. Wiedijk. 'Formal C Semantics: CompCert and the C Standard'. In: *International Conference on Interactive Theorem Proving*. Volume 8558. LNCS. 2014, pages 543–548. DOI: 10.1007/978-3-319-08970-6_36.

[62] R. Kumar, M. O. Myreen, M. Norrish and S. Owens. 'CakeML: A Verified Implementation of ML'. In: *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2014, pages 179–191. DOI: 10.1145/2535838.2535841.

[63] P. Lammich. 'Automatic Data Refinement'. In: *Proceedings of the 4th International Conference on Interactive Theorem Proving*. Volume 7998. LNCS. 2013, pages 84–99. DOI: 10.1007/978-3-642-39634-2_9.

[64] P. Lammich and T. Tuerk. 'Applying Data Refinement for Monadic Programs to Hopcroft's Algorithm'. In: *Interactive Theorem Proving*. Volume 7406. LNCS. 2012, pages 166–182. DOI: 10.1007/978-3-642-32347-8_12.

[65] D. Leinenbach, W. Paul and E. Petrova. 'Towards the Formal Verification of a Co Compiler: Code Generation and Implementation Correctness'. In: *Proceedings of the 3rd IEEE International Conference on Software Engineering and Formal Methods*. 2005, pages 2–12. DOI: 10.1109/SEFM.2005.51.

[66] D. Leinenbach and E. Petrova. 'Pervasive Compiler Verification—From Verified Programs to Verified Systems'. In: *Proceedings of the 3rd Systems Software Verification*. Volume 217. Electronic Notes in Theoretical Computer Science. 2008, pages 23–40. DOI: 10.1016/j.entcs.2008.06.040.

[67] X. Leroy. 'Formal verification of a realistic compiler'. In: *Communications of the ACM* 52.7 (2009), pages 107–115. DOI: 10.1145/1538788.1538814.

[68] G. Li. 'Validated Compilation through Logic'. In: *Proceedings of the 17th International Symposium on Formal Methods (FM)*. Volume 6664. LNCS. 2011, pages 169–183. DOI: 10.1007/978-3-642-21437-0_15.

[69] K. Mehlhorn and S. Näher. 'LEDA: A Platform for Combinational and Geometric Computing'. In: *Communications of the ACM* 38.1 (1994), pages 96–102. DOI: 10.1145/204865.204889.

[70] F. Mehta and T. Nipkow. 'Proving Pointer Programs in Higher-Order Logic'. In: *Proceedings of the 19th International Conference on Automated Deduction*. Volume 2741. LNCS. 2003, pages 121–135.

[71] L. M. de Moura and N. Bjørner. 'Z3: An Efficient SMT Solver'. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Volume 4963. LNCS. 2008, pages 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[72] Y. Moy. 'Automatic Modular Static Safety Checking for C Programs'. PhD thesis. Université Paris-Sud, 2009.

[73] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997, page 1004. ISBN: 978-1-55860-320-2.

[74] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao and G. Klein. 'seL4: from General Purpose to a Proof of Information Flow Enforcement'. In: *IEEE Symposium on Security and Privacy*. 2013, pages 415–429. DOI: 10.1109/SP.2013.35.

[75] M. O. Myreen. 'Functional Programs: Conversions between Deep and Shallow Embeddings'. English. In: *Proceedings of the 3rd International Conference on Interactive Theorem Proving*. Volume 7406. LNCS. 2012, pages 412–417. DOI: 10.1007/978-3-642-32347-8_29.

[76] M. O. Myreen, A. C. J. Fox and M. J. C. Gordon. 'Hoare Logic for ARM Machine Code'. In: *International Symposium on Fundamentals of Software Engineering*. Volume 4767. LNCS. 2007, pages 272–286. DOI: 10.1007/978-3-540-75698-9_18.

[77] M. O. Myreen, M. J. C. Gordon and K. Slind. 'Machine-code Verification for Multiple Architectures: An Application of Decompilation into Logic'. In: *Proceedings of the 2008 Conference on Formal Methods in Computer-Aided Design*. 2008. ISBN: 978-1-4244-2735-2.

[78] M. O. Myreen, M. J. C. Gordon and K. Slind. 'Decompilation into logic – Improved'. In: *Proceedings of the 2012 Conference on Formal Methods in Computer-Aided Design*. 2012, pages 78–81. ISBN: 978-1-4673-4832-4.

[79] M. O. Myreen and S. Owens. 'Proof-producing Synthesis of ML from Higher-order Logic'. In: *SIGPLAN Notices* 47.9 (2012), pages 115–126. DOI: 10.1145/2398856.2364545.

[80] NICTA Australia. *The eChronos Real-Time Operating System*. Accessed July 2014. 2014. URL: http://ssrg.nicta.com.au/projects/TS/echronos/.

[81] NICTA Australia, Trustworthy Systems Team. *seL4 Microkernel*. Accessed July 2014. 2014. DOI: 10.5281/zenodo.11247. URL: http://sel4.systems/.

[82] NICTA Australia, Trustworthy Systems Team. *seL4 Proofs v1.03*. Accessed August 2014. 2014. DOI: 10.5281/zenodo.11248. URL: http://sel4.systems/.

[83] T. Nipkow, L. Paulson and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Volume 2283. LNCS. Springer, 2002. DOI: 10.1007/3-540-45949-9.

[84] M. Norrish. 'C formalised in HOL'. PhD thesis. University of Cambridge Computer Laboratory, 1998.

[85] M. Norrish. *C-to-Isabelle Parser, version 1.13.0*. Accessed July 2014. 2013. URL: http://ertos.nicta.com.au/software/c-parser/.

[86] L. Noschinski, C. Rizkallah and K. Mehlhorn. 'Verification of Certifying Computations through AutoCorres and Simpl'. In: *NASA Formal Methods*. Volume 8430. LNCS. 2014, pages 46–61. DOI: 10.1007/978-3-319-06200-6_4.

[87] D. von Oheimb and T. Nipkow. 'Machine-Checking the Java Specification: Proving Type-Safety'. In: *Formal Syntax and Semantics of Java*. Volume 1523. LNCS. 1999, pages 119–156. DOI: 10.1007/3-540-48737-9_4.

[88] N. S. Papaspyrou. 'Denotational semantics of ANSI C'. In: *Computer Standards and Interfaces* 23.3 (2001), pages 169–185. DOI: 10.1016/S0920-5489(01)00059-9.

[89] J. C. Reynolds. 'Separation logic: A logic for shared mutable data structures'. In: *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE. 2002, pages 55–74. DOI: 10.1109/LICS.2002.1029817.

[90] D. M. Ritchie. 'The Development of the C Language'. In: *The Second ACM SIGPLAN Conference on History of Programming Languages*. HOPL-II. 1993, pages 201–208. DOI: 10.1145/154766.155580.

[91] N. Schirmer. 'A Verification Environment for Sequential Imperative Programs in Isabelle/HOL'. In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Volume 3452. LNCS. 2005, pages 398–414. DOI: 10.1007/978-3-540-32275-7_26.

[92] N. Schirmer. 'Verification of Sequential Imperative Programs in Isabelle/HOL'. PhD thesis. Technische Universität München, 2006.

[93] N. Schirmer. 'A Sequential Imperative Programming Language Syntax, Semantics, Hoare Logics and Verification Environment'. In: *Archive of Formal Proofs* (2008). Formal proof development. ISSN: 2150-914X. URL: http://afp.sf.net/entries/Simpl.shtml.

[94] H. Schorr and W. M. Waite. 'An efficient machine-independent procedure for garbage collection in various list structures'. In: *Communications of the ACM* 10.8 (1967), pages 501–506. DOI: 10.1145/363534.363554.

[95] T. Sewell, M. Myreen and G. Klein. 'Translation Validation for a Verified OS Kernel'. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2013, pages 471–481. DOI: 10.1145/2499370.2462183.

[96] T. Sewell, S. Winwood, P. Gammie, T. Murray, J. Andronick and G. Klein. 'seL4 Enforces Integrity'. In: *Proceedings of the 2nd International Conference on Interactive Theorem Proving*. Volume 6898. LNCS. 2011, pages 325–340. DOI: 10.1007/978-3-642-22863-6_24.

[97] N. Suzuki. *Automatic Verification of Programs with Complex Data Structures*. Outstanding Dissertations in the Computer Sciences Series. Garland Publishing, 1980. ISBN: 978-0-824-04425-1.

[98] *The Frama-C platform*. Accessed July 2014. 2008. URL: http://www.frama-c.cea.fr/.

[99] *The GNU Compiler Collection*. URL: http://gcc.gnu.org/.

[100] H. Tuch. 'Formal Memory Models for Verifying C Systems Code'. PhD thesis. UNSW, 2008.

[101] H. Tuch. 'Structured Types and Separation Logic'. In: *Proceedings of the 3rd Systems Software Verification*. Volume 217. Electronic Notes in Theoretical Computer Science. 2008, pages 41–59.

[102] H. Tuch. 'Formal verification of C systems code: Structured types, separation logic and theorem proving'. In: *Journal of Automated Reasoning: Special Issue on Operating System Verification* 42.2–4 (2009), pages 125–187.

[103] H. Tuch, G. Klein and M. Norrish. 'Types, Bytes, and Separation Logic'. In: *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2007, pages 97–108.

[104] P. Wadler. 'Monads for functional programming'. In: *Advanced Functional Programming*. Volume 925. LNCS. 1995, pages 24–52. DOI: 10.1007/3-540-59451-5_2.

[105] X. Wang, H. Chen, A. Cheung, Z. Jia, N. Zeldovich and M. F. Kaashoek. 'Undefined behavior: what happened to my code?' In: *Proceedings of the 3rd Asia-Pacific Workshop on Systems (APSys)*. 2012, 9:1–9:7. DOI: 10.1145/2349896.2349905.

[106] D. Wasserrab, T. Nipkow, G. Snelting and F. Tip. 'An operational semantics and type safety proof for multiple inheritance in C++'. In: *SIGPLAN Notices*. Volume 41. 10. ACM. 2006, pages 345–362. DOI: 10.1145/1167473.1167503.

[107] M. Wenzel. 'Isabelle/jEdit — A Prover IDE within the PIDE Framework'. In: *Intelligent Computer Mathematics*. Volume 7362. LNCS. 2012, pages 468–471. DOI: 10.1007/978-3-642-31374-5_38.

[108] M. Wenzel. 'Isabelle/Isar—a versatile environment for human-readable formal proof documents'. PhD thesis. Technische Universität München, 2002.

[109] E. D. Willink. 'Meta-Compilation for C++'. PhD thesis. University of Surrey, United Kingdom, 2001.

[110] S. Winwood, G. Klein, T. Sewell, J. Andronick, D. Cock and M. Norrish. 'Mind the Gap: A Verification Framework for Low-Level C'. In: *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*. Volume 5674. LNCS. 2009, pages 500–515. DOI: 10.1007/978-3-642-03359-9_34.

[111] X. Yin, J. C. Knight, E. A. Nguyen and W. Weimer. 'Formal Verification by Reverse Synthesis'. In: *Computer Safety, Reliability and Security*. Volume 5219. LNCS. 2008, pages 305–319. DOI: 10.1007/978-3-540-87698-4_26.