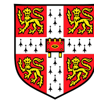
Technical Report

UCAM-CL-TR-453
ISSN 1476-2986

Number 453



UNIVERSITY OF CAMBRIDGE
Computer Laboratory

C formalised in HOL

Michael Norrish

December 1998

JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

© 1998 Michael Norrish

This technical report is based on a dissertation submitted by the author for the degree of Doctor of Philosophy to the University of Cambridge.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

Series editor: Markus Kuhn

ISSN 1476-2986

Abstract

We present a formal semantics of the C programming language, covering both the type system and the dynamic behaviour of programs. The semantics is wide-ranging, covering most of the language, with its most significant omission being the C library. Using a structural operational semantics we specify transition relations for C's expressions, statements and declarations in higher order logic.

The consistency of our definition is assured by its specification in the HOL theorem prover. With the theorem prover, we have used the semantics as the basis for a set of proofs of interesting theorems about C. We investigate properties of expressions and statements separately.

In our chapter of results about expressions, we begin with two results about the interaction between the type system and the dynamic semantics. We have both *type preservation*, that the values produced by expressions conform to the type predicted for them; and *type safety*, that typed expressions will not block, but will either evaluate to a value, or cause undefined behaviour. We then also show that two broad classes of expression are deterministic. This last result is of considerable practical value as it makes later verification proofs significantly easier.

In our chapter of results about statements, we prove a series of derived rules that provide C with Floyd-Hoare style "axiomatic" rules for verifying properties of programs. These rules are consequences of the original semantics, not independently stated axioms, so we can be sure of their soundness. This chapter also proves the correctness of an automatic tool for constructing post-conditions for loops with `break` and `return` statements.

Finally, we perform some simple verification case studies, going some way towards demonstrating practical utility for the semantics and accompanying tools.

This technical report is substantially the same as the PhD thesis I submitted in August 1998. The minor differences between that document and this are principally improvements suggested by my examiners Andy Gordon and Tom Melham, whom I thank for their help and careful reading.

Contents

1 Introduction **1**
1.1 General aims 1
1.2 Programming language formalisation 2
1.3 Theorem proving and mechanisation 5
1.4 Software verification 6
1.5 The C language 7
1.5.1 C terminology 8
1.6 Related work 9
1.7 Report structure 12

2 Statics **13**
2.1 The rôle of statics 13
2.2 C's types 16
2.3 Pointers and arrays 18
2.4 Types and the abstract machine 21
2.5 Expressions and their types 22
2.6 Statements 26
2.7 Mechanisation of the static semantics 28

3 Dynamics **33**
3.1 The program state 33
3.1.1 Accessing memory 34
3.1.2 Side effects 35
3.1.3 Notation—general observations 37
3.2 Undefinedness 39
3.3 Expression evaluation 41
3.3.1 Expression evaluation contexts 42
3.3.2 Base cases—values out of memory 43
3.3.3 Value producing operators 46
3.3.4 Side effect operators 49
3.3.5 Function calls—interfacing with statements 53

3.4 Statement execution 55
3.4.1 Simple statements 57
3.4.2 Interruptions 58
3.4.3 Compound statements 58
3.4.4 Conditional statements 59
3.4.5 Iteration 60
3.5 Variable declarations 62
3.6 Mechanisation 63

4 Expressions 67
4.1 Preliminaries 67
4.2 Symbolic evaluation and its pitfalls 71
4.3 Pure expressions 73
4.3.1 Proof outline 74
4.3.2 Pure expression determinism 75
4.4 Sequence point free expressions 77
4.4.1 A sequence point free diamond property 79
4.5 Undefined evaluations 83

5 Statements 87
5.1 A programming logic for C 87
5.2 Automatic loop exit analysis 93

6 Verification 101
6.1 Factorial 102
6.2 `strcpy` 104
6.3 The failed BDD example 108
6.4 Verification tools 110

7 Conclusion 113
7.1 Future work 115

A Definitions 119
A.1 Syntax 119
A.2 Semantics 124

B Theorems 139

Chapter 1

Introduction

We describe how programming language semantics can be formalised, emphasising the importance of the theorem prover in our work, and then briefly discuss software verification. We introduce some important C terminology and concepts, discuss other attempts to define C, and then lay out the structure of the rest of this work.

1.1 General aims

This report presents a formal definition of the programming language C. Both in avoiding errors made by others, and in formalising more of the language than has been done in the past, this work is a contribution to what might be called the "taxonomical" side of computer science: we give a precise and accurate description of one of the field's most famous creations. Moreover, because of the abstract mathematical nature of programming languages, we are able to describe C formally and rigorously.

The C programming language was created in the years 1969–1973, the result of work by Ken Thompson and Dennis Ritchie [Rit93] at the then Bell Telephone Laboratories. C's rôle was as the system implementation language for the new Unix operating system. Over the course of the next two decades, C spread widely, first on the heels of Unix, and then onto other systems as its suitability as a general purpose programming language became apparent. In 1989, C was given a rigorous definition by the ANSI standard [ANS89], which was in turn adopted as an ISO standard [ISO90] in 1990. The effort of standardisation was made "to promote portability, reliability, maintainability, and efficient execution of C language programs on a variety of computing systems" [ANS89, §*Abstract*], an effort necessary because of the danger of diverging language implementations blurring

exactly what was and was not a C program.

C is a very widely used language. It has been used to write operating systems such as Unix, OS/2, and Windows; applications such as Netscape and Microsoft Word; and it is used in fields from avionics to compiler construction. Its presence in today's operating systems has given the language a base from which it has expanded into many diverse areas of application development. So, while it is not necessarily of more intrinsic interest than more obscure languages, C is such an important part of the computer science environment that we feel the taxonomic motive for its formal study is a reasonable one.

C also combines a number of interesting features on the theoretical front, making it additionally interesting as a subject of study. For example, C's expressions both are side-effecting and have *very* under-specified evaluation orders. If these semantic features were the main area of interest in studying a language, then it would clearly be easier to construct a simple calculus that included these features and little else. (In this way, Milner's CCS [Mil89] and the object calculi of Abadi and Cardelli [AC96] present elegant approaches to core ideas behind concurrency and objects respectively.) However, we prefer to attack as much of C as possible all at once. As Milner and Tofte point out in the commentary on the definition of SML [MT90], this study of languages in their entirety has its own grounds for interest, and we further feel that our study of C gives us a possible application in the area of software verification.

This report thus has a second general aim: to build on the definition of the C semantics to develop theory and technology to enable verification of C programs. The remainder of this introductory chapter will discuss both these goals in more detail.

1.2 Programming language formalisation

We hold that the term "formal" implies "admits logical reasoning". We thus argue that C is not formally defined by the C standard [ISO90], because much (though not all) of that document relies on English language descriptions of program behaviour. Such descriptions, however carefully formulated, inevitably suffer from the ambiguities and obscurities inherent in natural language. An indication that the C standard is no more immune to this problem than any other natural language document is provided by the "Defect reports" [JTC], queries made of the standardisation committee asking for clarification of the standard's murkier details.

Those parts of the C standard that are formal are those that concern themselves with the definition of the language's syntax. Formally defining a language's syntax is a well-understood application of language theory. Moreover, the theory of automata long ago advanced to the point where the mechanical manipulation of grammars is common-place. Tools such as `yacc` and `lex` enable the automatic creation of programs to perform syntactic analysis of program source, thereby simplifying the process of compiler construction.

As C's syntax is already formally defined, we will not look at syntax any further. Our goal is to formalise what remains: the semantics of the language. Supplying programming languages with formal definitions is a major field of computer science, and it is not our aim here to provide an introduction to it. Instead we concentrate on describing our work in general terms, and placing it in the context of other work in the area of semantics.

We present C's semantics as a structural operational semantics. An operational semantics presents a formal description of an abstract machine that interprets the syntax of the language. The abstract machine's behaviour is then held to correspond to the behaviour one would expect when executing the program on an actual computer. This style of definition was used in the definition of Standard ML by Milner, Tofte and Harper [MTH90]. This example, one of the most famous formal language definitions, is a clear demonstration that a large language can be formalised in this manner. Moreover, operational semantics have proved relatively easy to mechanise as the technology for performing inductive definitions is well developed. As we shall later discuss, the semantics of SML itself has been one of the more tempting mechanisation "targets". A useful summary of other language formalisations, with varying degrees of formality, appears in VanInwegen's thesis [Van96].

One difference between our work and the definition of SML is that the latter is the first definition of the language. It constructs an abstract entity that (modulo any errors making the definition inconsistent, see Kahrs [Kah93]) is necessarily correct because it is *the* definition. We are not so lucky with our work on C. Here we must produce an accurate semantics for an existing language, one which has already been defined. An important question arising from this is whether or not our semantics conforms to the standard's definition.

Precisely because the standard's definition of C is not formal, we can never hope to *prove* our formal definition consistent with it. At best we can hope that our definition comes to be seen as correct by the community of people concerned with C's definition and standardisation. Such a

community can perform very useful error-checking.[1] In addition, if used as the basis for software tools that do not necessarily require a deep understanding of its details, a formal semantics may come to be accepted as correct simply because of what it has made possible in the pragmatic domain. Chapter four further discusses how our definition has passed various simple "sanity checks".

Alternatives to operational semantics include *denotational* and *axiomatic* semantics. A denotational semantics defines an appropriate mathematical space as a model for a language, and maps the language's syntax into that space in a way that is *compositional*. This property requires that the semantics of a syntactic phrase be a function of the semantics of the phrase's syntactic sub-components. While a denotational semantics for C is conceivable, it would be complicated both to devise the appropriate space for the semantic model, and to then ensure that one's denotation function was both correct and compositional. The mathematical theory of domains required for the treatment of unbounded loops and recursion has been handled in theorem provers before (notably in LCF [GMW79]), but its complications don't seem to be as well-served by the existing tools as the simpler demands made by operational semantics.

An axiomatic semantics for a language presents a set of rules that allow one to conclude that particular syntactic phrases will satisfy a specification, given as a precondition and a post-condition. Being inductively defined, axiomatic semantics share certain similarities with operational semantics. However, where an operational semantics presents an abstract machine whose evolving behaviour can be readily appreciated, an axiomatic semantics makes statements about program behaviour that can be rather harder to evaluate. A typical conclusion in an axiomatic semantics will be that if one starts program S in a state satisfying P, then the program will finish in a state where Q is true. This high descriptive level is very useful when a semantics comes to be used in an application such as software verification, but it is harder to confirm that such a definition corresponds to one's intuition. In an operational semantics, any given rule will more obviously express a transformation on a particular program state. In chapter 5, we shall link our operational definition with some rules written in an axiomatic style, thereby combining the confidence in our operational definition with the utility of axiomatic rules.

[1] Inasmuch as this community is present in the Usenet newsgroup `comp.std.c`, it has already had a much-appreciated influence on this report—the typically informed posts in this newsgroup, some of them responses to my own queries, often served to point out areas where our semantics might be incorrect, and thus proved an ongoing test for it as it developed.

1.3 Theorem proving and mechanisation

It is central to our thesis that the semantics of C is so complicated that it can only be usefully manipulated in the context of a *theorem prover*. In this environment, logical definitions can be made and theorems proved in a reliably sound way. Without mechanical support, reasoning with a big semantics is error-prone, and it can be hard to be confident that one's proofs are actually correct. For example, Syme recently worked on mechanising a proof of Java type soundness [Sym97b] that had originally been done by Drossopolou and Eisenbach [DE97]. Though their proof was substantially correct, Syme's mechanisation did find what he describes as "one major error and one major omission" in their proof.

Using a theorem prover means that we are confident that all of the results we have proved are correct. Having used the theorem prover HOL [GM93], we are particularly confident, as this system, following the example of its ancestor system LCF [GMW79], uses the strong type system of ML to guarantee that values of type theorem are only produced in ways that are logically sound. This guarantee extends to HOL's use of conservative principles of definition, meaning that we are not forced to assert definitions as potentially inconsistent axioms. Just as novice computer users are often told to type what they like, because nothing they can do can actually harm the computer, so too, novice users of HOL can proceed, knowing that whatever they do will not compromise the validity of the theorems they prove. This is quite a liberating sensation.

The foregoing is not meant to suggest that errors are impossible in HOL, or theorem proving systems in general. Two sorts of errors are relatively common. Firstly, one's definitions may not say what one thinks they do. Whether through simple typographical error or confusion at a higher level, an incorrect definition may prove the basis for the proof of a slew of results. Then, suddenly the edifice one is constructing collapses because a foundation stone is slightly out of place, and this prevents the proof of a desired property. We will discuss this sort of error further in chapters three and four. The second sort of error comes in proving results that are not useful. These theorems may take considerable effort to prove, but when they come to be applied, it is found that the hard-won theorem doesn't pertain to the situations it was intended for. Just as typed programming languages can not prevent errors in code, HOL's guarantees of soundness only forestall certain (but important!) kinds of theorem proving errors.

Our mechanisation of C is called Cholera.

1.4 Software verification

Generally speaking, *software verification* is the task of assuring oneself that a piece of software behaves in a particular way. Furthermore, one requires that this conclusion be backed up by assurances that are formal in nature. It is not enough to look at a piece of code and say to oneself "*Yes, this loop will terminate because my co-worker always writes such nice code.*" Instead such a conclusion (about the termination of the loop, rather than the niceness of the co-worker's code!) should be reached only after the code has been subject to some form of formal analysis.

But is this a reasonable thing to wish on any putative software verifier? In their famous criticism of software verification [MLP79], De Millo, Lipton and Perlis present a strong case to the effect that even if someone were willing to subject themselves to the unimaginable demands of producing such a thing, a verification could have no real value because its mind-numbing complexity would not be usefully communicable to anybody else. And where is the value of a verification if no-one else will believe it?

Though we believe that De Millo, Lipton and Perlis elsewhere over-state their case as to the impossibility of producing valid verifications, their point about the power of the social process is a good one. If other humans won't accept our verifications, it seems that we must substitute a computer instead. A mechanical theorem prover will not give us the brush-off when we force it to read over our verification proofs. Moreover, once mechanical verification systems have earned the trust of the human community, an individual's verification can come to be trusted if it has been certified correct by the machine.

An intriguing alternative is suggested by George Necula's work on *proof-carrying code* [Nec97]. In this scheme, humans are never required to blindly trust that someone else's verification of someone else's code is correct. Instead humans trust their computer's mechanical proof-checker: it confirms that potentially suspicious code behaves as it should (or, alternatively, doesn't behave as it shouldn't), by checking the verification proof that accompanies the code. This is an appealing proposal because humans then never have to "consume" proofs themselves. Given their complex and generally incomprehensible nature, it seems entirely appropriate that once produced (presumably by a human-machine combination), such proofs should only ever be consumed by machines.

We should also consider the case De Millo, Lipton and Perlis make to the effect that software verification is a practical impossibility. This criticism is one that we answer directly, at least to some extent, in the presented case study examples (chapter 6). We also hope that our definition of C's se-

mantics and the derivation of consequences from this semantics will prove useful enough to allow others to answer the question of whether or not software verification is ever really to succeed.

Software verification is itself just one possible approach to the goal of producing reliable software. Another approach, which is much further down the track towards pragmatic acceptability, is that exemplified by the LCLint static checker [EGHT94, Eva96]. This work provides an appealing tool that checks C programs for errors in conjunction with user-provided specifications of the properties individual functions are supposed to respect. Though the more recent of the two papers referenced above admits that the tool is both unsound and incomplete, the cited experience seems to suggest that LCLint is nonetheless a very useful tool. Our (contrasting) approach is to make correctness an overriding concern, and hope that future work will derive pragmatically useful tools on top of our sound foundations.

1.5 The C language

C is a third generation imperative language in the tradition of Algol. It is a typed language, and its types can be extended by the construction of new types from old types in a regular way. For example, from any type τ, one can construct types corresponding to “pointer-to-τ” and “array-of-τ”. C’s statements are unremarkable but provide standard looping and conditional constructs such as `while`, `for`, `if` and case switching.

It is principally with its expressions that C gains its special flavour. It is here that C’s low-level nature is most apparent. C encourages the programmer to think of values as sequences of bits by providing a comprehensive set of bit-level operators such as and (`&`), or (`|`) and xor (`^`). C also makes pointers and variable addresses a fundamental part of the language semantics. Arrays are manipulated through pointers, and reference parameter passing is accomplished by explicitly requiring the user to pass the addresses of variables. Furthermore, the language allows all values to be accessed a byte at a time, ignoring any higher level structure that has been imposed on those values. The type system has perhaps its biggest hole in variadic functions. Not only must the program dynamically establish the type of the trailing arguments to such functions, which may all be different, but it must also determine when these arguments have been exhausted. All these factors make it hard for programmers to escape the feeling that they are very close to the “bare metal” of their machine.

C does not support programming in the large particularly well. At the global level, the name-space for both function and variable identifiers is

entirely flat. Each C source code file (a *translation unit*) has its own internal name-space, which can keep names hidden, but if names are allowed to escape this, then they are visible everywhere. Variables can be local to functions also, whether freshly created with each invocation, or keeping their value from call to call. Functions do not nest, however.

It is in this context that the distinction between variable and function *definition* and *declaration* is most important. A declaration is a non-committal statement to the effect that a variable or function exists, but in the case of functions, is not a definition of that function, and in the case of variables does not allocate the necessary space in memory for it. Declarations are used to import names from other modules in the knowledge that a linking phase will resolve the issue of where the object (function or variable) behind the name is. A definition is a declaration, and also provides either a function body or causes the setting aside of memory for the allocation of a variable.

1.5.1 C terminology

The C community tends to use a language of its own to describe notions relating to its language's semantics. Here we introduce some of the more important terms (*c.f.* the glossary in [ANS89, §1.6]). Firstly there are three distinct ways in which the standard under-specifies execution of C programs. We explain each and describe in general how they are modelled in the formal semantics.

Implementation-defined: Implementation-defined constructs are those which must have a definite meaning, but for which the standard has passed responsibility to the implementation. The implementation is required to document its choice of meaning. An example is the byte-ordering within multi-byte numeric objects (big-endian *vs* little-endian).

We handle implementation-defined behaviour by defining, but under-specifying some constant in the semantics. In this way, we model the fact that there is *a* well-defined behaviour, but make it impossible for a user to rely on it being a *particular* behaviour. For example, `INT_MAX`, the largest value that can be stored in the `int` type must be at least 32767, but can be more.

Unspecified: A construct or program form for which behaviour is unspecified is one where the standard imposes no requirement. For example,

the order of evaluation of expressions is unspecified. Here an implementation need not document its behaviour, and thus may choose to do different things in quite similar, if not identical, situations.

We handle unspecified behaviours by always allowing all possible behaviours. In the case of expression evaluation, all possible evaluation orders can arise. One can not then claim that a program's behaviour will have a particular result without confirming that all possible behaviours lead to the same result.

Undefined: Undefined behaviour results when a program attempts to do something which is semantically illegal. For example, undefined behaviour occurs when uninitialised memory is accessed, when a null pointer is dereferenced, or when a side effect attempts to update a memory object which has already been accessed in the same phase of expression evaluation. We treat all such behaviour as equivalent to a transition into a special state where no further action takes place. In implementations, a program which attempts an undefined behaviour will in all likelihood do something, and this something may in fact be quite reasonable. Nonetheless, there is no way of relying on undefined behaviour to do anything in particular, so our approach of effectively aborting the abstract machine as soon as undefined behaviour occurs is safest. To do anything else would be to suggest that a particular behaviour could be relied upon, and this would necessarily be erroneous.

Related to the above notions is that of being a *strictly conforming* program. Such a program's behaviour doesn't depend on any implementation-defined, unspecified or undefined behaviour. Though strictly conforming C programs are the ideal, it is still the case that there are legal programs that are not strictly conforming. For example, there are legal programs that behave non-deterministically, and the semantics as presented here and in the standard reflects this.

1.6 Related work

There are a number of attempts to define C's semantics extant in the literature. Here we briefly summarise those definitions that we are aware of. Our own work goes beyond those presented here in covering more of the language and covering what it does more faithfully to the standard.

For example, our presentation of C's semantics (correctly) insists that the expression

```
(x = 0) + (x = 0)
```

is undefined, and thus illegal. This identification is one that distinguishes our formal semantics from all the others so far published. Our definition of the semantics also covers more of the language than other work to date. For example, we handle function pointers and structure values returned from functions, which are both omitted in the other work.

The work closest to our own in approach is that done by Cook and Subramanian [SC96]. Like our own, this work explicitly has as its goal the verification of C programs, and bases itself in a mechanised theorem prover. Cook and Subramanian's work defines a C interpreter as a Lisp function in the Nqthm theorem prover, and then proves properties of this interpreter using the Nqthm logic (quantifier-free first order logic with equality and induction principles). This approach leads to a semantics which is automatically executable, a feature which is a great help in validating one's semantics. Cook and Subramanian's work mechanises quite a small C subset, using a restricted type system and simplifying the allowed expressions. They also omit all unspecified aspects of the language by making choices such as evaluating expressions left-to-right, and choosing specific values for constants such as the number of bytes in a word. Nonetheless, Cook and Subramanian were able to verify a factorial program, and suggest in [SC96] that it should be feasible to build a system for verifying straightline C programs automatically.

Paul Black's work [BW96, BW98] is another approach to verification of C in a mechanised context. While our approach is to use an operational semantics, Black focuses on giving an axiomatic semantics for C, and then using this to verify examples of C code, in particular a simple World-wide Web server. However, in the search for a pragmatic tool, Black has adopted an extra-logical approach to expressions. Rather than giving them a rigorous semantics, he instead assumes the presence of a semantic equivalence relation with which to rewrite difficult expressions (such as those involving side effects) into more tractable forms. We agree with Black's conclusion in [BW98] that his system will need to be based on a lower-level definition of C's semantics if it is to be judged properly reliable.

Characterising Black's work as a high level approach to the problem in search of a basis in low level reliability, we claim that our own work is a low level approach that attempts to derive high level consequences of the sort asserted by Black as axiomatic. Our later "meta-theory" chapters 4 and 5 explore how our basic definition can be extended with high level

consequences. Perhaps future work will see our work and Black's start to "meet in the middle".

More recently, Mark Bofinger has published a definition of C's semantics [Bof98], again with the aim of providing a basis for performing program verification. This work differs from that previously discussed and our own in not having been done in a theorem-prover, or with the help of any automated support. After definition, Bofinger's semantics is then used to support the development of a refinement formalism for turning specifications into C code (including an example refinement of a number of variations on the factorial function).

Bofinger's semantics attempts to capture strict conformance by using rules expressing the notion of "evaluating to the same result regardless of the order of evaluation chosen". However, because this semantics uses a "big-step" operational semantics, it can not express all of the ways in which C expressions might evaluate. For example, in an expression $(a+b)+(c+d)$, Bofinger's semantics will allow for the following eight possible orders: $abcd$, $bacd$, $abdc$, $badc$, $cdab$, $cdba$, $dcab$, and $dcba$. However, it does not examine order $acdb$, one of many other orders which *are* permitted to occur.

Further, Bofinger's semantics does not include the post-increment and post-decrement operators. The latter are excluded because of a misapprehension about the way in which side effects are applied. Bofinger correctly states that side effects from these operators may be applied at any time, but fails to realise that this is also true of side effects that arise from assignment expressions as well.

The semantics given by Gurevich and Huggins [GH93] using the evolving algebras formalism [Gur91] models expression evaluation in a similar way to Bofinger, thereby missing possible orders of evaluation. It also does not correctly model the way in which multiple side effects may give rise to undefined behaviour, as in the distinguishing example above. However, the evolving algebra approach does seem quite expressive. The formalism seems to allow a pleasant layering in the description of the semantics—the semantics is presented quite abstractly initially, with details filled in over a number of stages. This semantics was also developed without mechanical assistance.

Finally, Lars Andersen gives a semantics for much of C as part of his development of a partial evaluation system for the language [And94]. He finesses the issue of different evaluation orders by stating that his semantics is only for strictly conforming programs, where differing evaluation orders can not make a difference. His semantics is thus able to choose an evaluation order (left to right) and also chooses to apply side effects as soon as they are generated. This approach will not give incorrect results for cor-

rect programs, but it will incorrectly give a meaning to incorrect programs. While his approach has obvious pragmatic advantages, it is not strictly a description of C but rather a formal description of a possible C implementation.

1.7 Report structure

Chapters two and three of this report present the formal definition of a substantial subset of the C language. Each of these chapters is a rigorous and correct description, but in some respects does not quite correspond to what was done in the Cholera mechanisation. For this reason, these semantics chapters both include concluding sections that describe how the mechanisation in the theorem prover differs from the semantics as given.

Chapters four and five describe the principal results that we can prove as a consequence of the language definition. Chapter four describes expression evaluation, while Chapter five describes statement evaluation. Results include subject reduction and type safety properties for expression evaluation, and the simple beginnings of a programming logic for reasoning at the statement level. In both chapters, the results stated are those that have been proved in the theorem prover.

Chapter six describes a series of case study verifications. This includes a description both of what was proven, and how this was accomplished in terms of use of previously proved results and also in the use of bespoke theorem proving technology. Chapter seven concludes. Finally, we provide an appendix of extracts from the Cholera source code. These are presented to give a picture of some of the mechanisation's nastier details. These extracts include the full text of the theorems that are stated in chapters four and five, and a selection of definitions, again as they were seen by the theorem prover.

I am determinéd to prove a villain
And hate the idle pleasures of these days.
Plots have I laid, inductions dangerous...
(*Richard III*, Act 1, Scene 1)

Chapter 2

Statics

We describe C's type system, discussing first the nature of type systems in general and then specifying the forms of valid types in C. We then explain the often misunderstood relation between pointer and array types in C. The nature of C's abstract machine leads us into an investigation of further properties of types and values, before we describe the assignment of types to expressions. After briefly looking at the various statement and declaration forms, we finish with a description of the mechanisation of the static semantics in the theorem prover.

2.1 The rôle of statics

A static semantics for a programming language is a classification of that language's syntactic phrases into *types*. Thus, static semantics are also known as "type systems". Naïvely, types can be seen as corresponding to sets of values, and a static semantics predicts the type of the value denoted by a syntactic phrase. Alternatively, a static semantics can be seen as a low-powered "meaning ascriber". Where a dynamic semantics gives the meaning of $3 + 5$ as 8, the static semantics might only claim that $3 + 5$ denotes a value in the set of natural numbers. In this light, static and dynamic semantics can be seen to fall onto the same spectrum. Both are functions from syntactic values to sets of semantic values. Naturally, a dynamic semantics should yield singleton sets.[1]

Despite this identification at one level, a static semantics traditionally embodies certain features that set it apart from dynamic semantics. In programming languages that include state, an easy distinction to draw is that

[1] In the case where a semantics is non-deterministic, the semantic values might themselves be sets, but these still represent one value at the level of the "meaning ascribers".

a static semantics will not consider the state's possible value in characterising the meaning of an expression. It is state-independent in a way that the dynamic semantics can not be. For example, the static semantics of such a language will give the meaning of a variable reference the (declared) type of that variable, while the dynamic semantics will give the meaning as a function which accesses the state and yields the actual value.

More significantly, a static semantics is typically allowed the privilege of being vague about exceptional conditions. In particular, static semantics are allowed to hedge their bets by assigning types on the assumption that the expression in question does not do anything "questionable". In the alternative view of semantics as functions from syntax to sets of semantic values, static semantics are allowed to extend the result set with exceptional or error values. For example, the SML static semantics gives the type of

```
fn x => x div 0
```

as `int -> int`. In this example the static semantics deduces the type of `x div 0` to be `int`, despite the fact that it will necessarily raise an exception. Its prediction of `int` as the type of this expression has to be read as really meaning the set of all possible `int` values, in addition to all the exceptions possibly generated by `div`.

Finally, the most important use of static semantics is probably that of limiting the language over which the dynamic semantics is expected to operate. In this role, a static semantics limits programs to those which are "well-typed". If a variable of type `int` is assigned a value of type string in the program text, an implementation can use a static semantics to reject this program, and thus avoid defining a meaning for this form in the dynamic semantics. Programming systems where this discipline is imposed are typically said to support "type-checking at compile time", while those that dispense with a static semantics have at best "run-time" or "dynamic" type-checking.

If used in this way, a static semantics provides languages with a way of achieving a separation of concerns. The static semantics filters out evidently meaningless programs, allowing the dynamic semantics to assume that its programs are all well-typed. In an ideal world, a language's static type system will only give types to programs with meanings. Unfortunately, languages differ in the way they handle run-time errors. In a language such as SML, we might reasonably consider division by zero to be an error, but if this occurs during the execution of a program, the dynamic semantics specifies that an exception must be raised. By giving even "erroneous" constructs a defined behaviour, the language accepts them as legitimate,

allowing us to reason about them.

In C, by contrast, erroneous run-time situations really are errors, and the standard imposes no requirements on the semantics. Instead, errors simply result in "undefined behaviour". This is very unfortunate, because it means that in practice it is extremely difficult to tell if a C program is behaving correctly. Rather than actually being correct, a C program may just happen to be exhibiting correct behaviour because of the quirks of the particular implementation being used. In the terminology of Cardelli [Car97], all of C's runtime errors are *untrapped*.

One might initially see this failing as one of the static semantics. One might hold that the situation is simply that the dynamic semantics is being expected to ascribe meanings to erroneous programs, and that these erroneous programs should have been filtered out by the static semantics. However, this is not reasonable. Rice's Theorem states that there can not be a general algorithm to distinguish non-trivial (i.e., non-empty or non-universal) subsets of recursive programs. Clearly that subset of programs whose members will exhibit some form of run-time error is non-trivial in this sense, so it is impossible to construct a decidable type system which will prevent division from zero occurring in the dynamic semantics.

C's failing is that its dynamic semantics is not sophisticated enough to trap the bulk of its possible runtime errors. Alternatively, if one allows that trapped exceptions aren't really errors at all (inasmuch as the semantics requires a behaviour for the situation), then C's problem is simply that it has run-time errors. As we shall see, C's type system is sound (it satisfies the property of *type preservation*), but this fact seems of little consequence in the face of these run-time errors.

This issue arises in [SV98], where Smith and Volpano discuss a simple variant of C with parametric polymorphism à la SML. Their language also has a sound type system, but still admits the possibility of run-time error. Detecting these errors at run-time would make the language implementation's *safe* (again Cardelli's [Car97] terminology), but would also compromise efficiency. In the same way, C implementations *could* detect all the various errors possible, but only a relatively small number of special purpose C implementations attempt to notice some run-time errors as they occur (Jones's and Kelly's bounds-checking implementation of `gcc` [JK95], for example), and none catch all possible errors.

2.2 C's types

C's type system is quite simple. A number of simple types with obvious connections to typical machine hardware are used as the basis for a system that allows the construction of new types from old. The "constructors" allow the generation of function, pointer, array and record (known in C as `struct`) types. First we introduce the following abstract syntax for types:

$$
\begin{array}{lcl}
i & ::= & \texttt{char} \mid \texttt{short} \mid \texttt{int} \mid \texttt{long} \\
\tau & ::= & \texttt{void} \mid \texttt{signed}\, i \mid \texttt{unsigned}\, i \mid \texttt{float} \mid \texttt{double} \mid \texttt{long double} \mid \\
 & & \tau\texttt{*} \mid \tau\texttt{[}n\texttt{]} \mid \texttt{struct}\, s \mid \tau_1 \times \tau_2 \times \ldots \tau_n \to \tau
\end{array}
$$

The `signed` and `unsigned` types are known as the *integral* types. Together with the floating point types (`float`, `double` and `long double`), these form the *arithmetic* types. In the concrete syntax, one can omit the `signed` keyword for all but the `char` type. We can thus write `int` instead of `signed int`. However, it is implementation-defined whether the `char` abbreviation stands for `unsigned` or `signed char`.

The `void` type fills two rôles in C; it serves as the return type for functions that do not return values (thus being similar to SML's `unit` type), and secondly, the type of pointers to `void` is a generic pointer type, that can point to any sort of address, and which is the basis for a primitive form of polymorphism. The type τ`*` is the type of pointers to τ, while the type τ`[`n`]` is that of an array of size n containing objects of type τ. Record types can't be denoted directly in this presentation as it is legitimate for types to be recursive by including pointers to `struct`s inside those `struct`s themselves. Here, we achieve the recursive "tying of the knot" by using a structure information environment (which we will denote using Σ, Σ_1 etc), a finite map from tags (denoted by s above) to finite sequences of pairs of names and types, the `struct`'s *members* or fields.

We omit the following features of C's type system: enumeration types, the type qualifiers `const` and `volatile`, bit-fields, and `union` types. Furthermore we do not allow functions to take variable numbers of arguments, and we also gloss over the `typedef` construct, assuming that this last facility is compiled out in such a way that occurrences of type identifiers are replaced with the type that they abbreviate.

Our treatment of functions and array types as function parameters is different from that in the standard. In the case of functions, the standard makes use of what it terms "pointers to functions". These are essentially variables that can contain references to functions, where possible values are all of the program's defined functions. This is how we shall treat func-

tion references henceforth, stripping them of the confusing semantic baggage associated with pointers. (The language in the standard is continually having to make exceptions for pointers to functions in its description of operations on pointers. It is not possible to perform pointer arithmetic on function references, and dereferencing of function "pointers" is an idempotent operation.) This clarity of exposition will also be evident in the discussion of the dynamic semantics. We will discuss the nature of array types, and how our definition differs from that given in the standard in section 2.3 below.

It can be seen that all of C's types correspond to finite sets of values, and we will occasionally abuse our notation to let a type name stand for the set of possible values that inhabit that type.

Even simplified in this way, the abstract syntax above permits illegal types. For example, it is illegal to have functions returning arrays, and arrays must have a positive size. The following definition of type well-formedness defines the legal types. We will write $\Sigma, \Theta \vdash \mathit{wf}(\tau)$ to mean that type τ is well-formed in structure information context Σ, assuming that all pointers to structures with tags in the set Θ are also well-formed. (The Θ parameter is necessary to allow `structs` to include pointers to themselves.) Furthermore, we will write $\Sigma \vdash \mathit{wf}(\tau)$ when we have Θ equal to the empty set, and this will be our usual meaning when we use the term "well-formed".

First, all of the basic types are well-formed:

$$\frac{}{\Sigma, \Theta \vdash \mathit{wf}(\texttt{signed}\ i)} \quad \frac{}{\Sigma, \Theta \vdash \mathit{wf}(\texttt{unsigned}\ i)} \quad \frac{}{\Sigma, \Theta \vdash \mathit{wf}(\texttt{void})}$$

$$\frac{}{\Sigma, \Theta \vdash \mathit{wf}(\texttt{float})} \quad \frac{}{\Sigma, \Theta \vdash \mathit{wf}(\texttt{double})} \quad \frac{}{\Sigma, \Theta \vdash \mathit{wf}(\texttt{long double})}$$

A pointer type is well-formed if the type it points at is well-formed, or if the tag of the structure type it's pointing at is assumed to be well-formed.

$$\frac{\Sigma, \Theta \vdash \mathit{wf}(\tau)}{\Sigma, \Theta \vdash \mathit{wf}(\tau *)} \qquad \frac{s \in \Theta}{\Sigma, \Theta \vdash \mathit{wf}(\texttt{struct}\ s *)}$$

Array types are well-formed if their base type is well-formed and not `void`, and if the number of elements is greater than zero.

$$\frac{\Sigma, \Theta \vdash \mathit{wf}(\tau) \quad \tau \neq \texttt{void} \quad n > 0}{\Sigma, \Theta \vdash \mathit{wf}(\tau\,\texttt{[}n\texttt{]})}$$

Function types are well-formed if all of their argument types are well-formed, non-`void` and not array types, and if their return type is well-formed and not an array type. Note that the number of arguments may

be zero, in which case the function type is similar to the type of those functions in SML that have argument type `unit`.

$$\frac{\Sigma, \Theta \vdash \mathit{wf}(\tau) \qquad \tau \text{ not an array type} \qquad \forall i.1 \leq i \leq n \Rightarrow \Sigma, \Theta \vdash \mathit{wf}(\tau_i) \wedge (\tau_i \neq \texttt{void}) \wedge \tau_i \text{ not an array type}}{\Sigma, \Theta \vdash \mathit{wf}(\tau_1 \times \tau_2 \times \ldots \tau_n \rightarrow \tau)}$$

A `struct` type is well-formed if all of its member types are well-formed and non-`void`, and if all of the member names are distinct. Further, we can assume that pointers to the `struct` in question are well-formed while examining the well-formedness of its members. Finally, `struct` types with no members are forbidden. Recall that $\Sigma(s)$ denotes the sequence of member names and types associated with `struct` tag s. We will write $\Sigma^1(s)$ to denote the sequence of the `struct`'s member names, and $\Sigma^2(s)$ to denote the sequence of the member types. We also overload $\in$ to mean sequence or list membership as well as set membership.

$$\frac{\Sigma(s) \neq \langle\rangle \qquad \Sigma^1(s) \text{ contains no duplicates} \qquad \forall \tau \in \Sigma^2(s).\ \Sigma, \Theta \cup \{s\} \vdash \mathit{wf}(\tau) \wedge (\tau \neq \texttt{void})}{\Sigma, \Theta \vdash \mathit{wf}(\texttt{struct}\ s)}$$

Our notion of well-formedness does not presuppose any particular method for specifying types in the language's concrete syntax. For example, C's concrete syntax defines a notion of "incomplete type" that corresponds to types which have not yet been fully defined. However, this is an artifact of what happens when a source file is translated, and has no place in the semantics.

Henceforth our meta-variables τ, τ_1 etc. will range over only well-formed types.

2.3 Pointers and arrays

C's pointers and arrays are often confused. Folk wisdom has it that "in C, arrays are just pointers". This is not true, though it is easy to see why the misunderstanding arises. Instead, it is more accurate to state that arrays are second-class types, that it is impossible to manipulate array values directly, and that arrays only exist in memory, not as values independent of location. Arrays can not be passed to functions, nor returned from them, and it is not possible to assign to variables that have array types. Moreover, there is no way of writing anonymous array values (except in the uninteresting case

where that value is used only to initialise an array that is being declared). There is no compelling design reason for this state of affairs, but it is a natural consequence of C's historical roots in the programming languages BCPL and B [Rit93]. (Indeed, `structs` originally suffered from the same sorts of restrictions as arrays.)

As memory-bound entities, it is natural that arrays *can* be manipulated with pointers. Because one can take the address of arbitrary objects in C, and because pointers support arithmetic, the elements of an array are all individually accessible once one has a pointer to any one of them. Supporting this style of use, array variables in C expressions do not yield a value corresponding to the vector of values that they consist of, but rather a value that is a pointer to the array's first element.

If `array` is a variable declared as an array of some sort, it is then legal to write both `array + i` and `*array` (where $*e$ denotes the dereferencing of a pointer value). Further, the syntax `array[i]` is not an independent array indexing operation, but is rather syntactic sugar for `*(array + i)`. While `array` is not a pointer, it does yield a pointer value in those contexts which require values. Values are also required in the context of parameters to functions, so that arrays passed as such are also converted to pointer values. (Naturally, this process of conversion, while significant semantically, is a "no-op" as far as any realistic implementation is concerned.)

The only context where an array variable does not decay into a pointer value is when it is an argument to the `sizeof` operator. The `sizeof` operator returns its argument's size in bytes. It is precisely because `sizeof` doesn't evaluate its argument in any real sense that this is acceptable. In fact, `sizeof` is again likely to be a no-op as far as a realistic implementation is concerned because it will be "compiled out" (this being a form of constant folding) when the program is first parsed.

In order to ensure that array values are never manipulated, there is one further exception made to what one might expect to be the language's general rules. Although there is no way of writing anonymous `struct` values in C, such values can be returned from functions. If such a `struct` has an array as one of its members, then one might imagine that member selection (the "`.`" operator) would give one access to a genuine array value that was not connected to some part of memory. As the rules for expression typing in section 2.5 will show, this possibility is expressly forbidden.

This manipulation of arrays through pointers is a natural idiom at the level of machine code, and its translation into a higher level language is often cited as proof of C's inadequacies. On the other hand, C's simple-minded type system does at least allow functions to be written which manipulate arrays of any size; something that contemporaneous Pascal did not

originally allow.

Moreover, C allows functions which require arrays of specified lengths as parameters. This is achieved by passing an array's address, using the address-of (&) operator. In terms of our "realistic implementation", this is a peculiar operation because it will involve passing an address which is the same as the address of that array's first element, albeit of a different type. The following two functions illustrate this:[2]

```
int f1(int *intptr) { return intptr[3]; }
int f2(int (*arrayptr)[20]) { return (*arrayptr)[3]; }
```

While `f1` takes a pointer to an integer as its parameter, `f2` takes a pointer to an array of 20 integers. If `a` were declared as such an array, `f1(a)` would achieve exactly the same effects as `f2(&a)`. However, if `a` were an array of 30 integers, `f2(&a)` would fail to type, while `f1(a)` would work just as before.

Finally, it is worth noting that the standard's presentation of the language allows for *incomplete* array types, i.e., array types without a specified size, written τ`[]`. In the context of declarations, these uses are a syntactic convenience, allowing the user to state that an array exists without having to repeat size information that is present where the array is defined. (This is of particular use in situations where the array in question is defined in another file entirely.) This use of incomplete arrays can be disregarded in our setting, where we are interested only in the final result of parsing, not the syntactic devices that make it possible.

However, C also allows pointers to incomplete array types to be passed to functions. At first sight it appears that our formalisation will have to allow this possibility as well. However, in the interests of simplicity we forbid this, noting that functions with such parameters are completely isomorphic statically and dynamically to functions that simply take pointers to the item type of the array. At the point of call, the value passed is exactly the same except that it must be cast to a different pointer type. When a pointer to a complete array type is passed, type-checking may preclude certain arguments, and the parameter may have the `sizeof` operator applied to it. Neither of these possibilities obtains when the array type is incomplete.

[2]Each function returns the fourth element of the array. This array is implicit in the case of `f1`. Incidentally, it should be clear by now why C's array indexing starts at zero, and not one.

2.4 Types and the abstract machine

The values that inhabit C's non-`void` types are not atomic. All can be seen as finite sequences of bytes in memory. This is not just a reflection of likely implementation strategies but is a requirement imposed by the language semantics. One expects that the objects that make up arrays and `struct`s should be individually accessible, and this in itself is a clear justification for being able to look at the same part of memory in different ways.

For example, if one passes the address of a `struct` to a function which changes one of the `struct`'s members, this change must also be visible in the dereferencing of a pointer to that member. Therefore, the abstract machine's model of memory must support the possibility of two values of different type in memory sharing the same address. Furthermore, the standard is clear that all address values (i.e., those values that inhabit pointer types) can be converted to pointers to character types. It is also explicitly permitted to alias values in memory by accessing them as characters, regardless of their original type. Thus it is clear that, at the very least, memory must be modelled as a map from addresses to character values. Characters are also known as *bytes* and as some possible byte values may not actually correspond to anything printable, or to any particular glyph, the term byte is to be preferred when discussing the contents of memory.[3]

The standard requires that every value in a given type take up the same amount of space in memory. (It is hard to imagine how space could be allocated for objects when they were declared otherwise.) Further, the length of signed and unsigned values of the same integral type must be the same. The length of this sequence for each type is known as its size. We will write $|\tau|$ to denote this size for various τ. Naturally, the size of the character types is exactly one, given their direct correspondence with bytes, but the sizes of other types vary from implementation to implementation.

This variation arises because of three reasons: firstly, the standard requires that the various numeric types support minimum ranges, but implementations may choose to support wider ranges. The `signed int` type must include all values in the range -32767 to 32767 for example, but often contains 32 bits' worth of values (twice as many bits' worth) on modern architectures. Secondly, possible variation in the number of bits in a byte affects the number of bytes required to store a given range of values. Finally, with the exception of the `unsigned char` type, it is permitted for certain bit sequences to fail to correspond to actual values in the given type. This may

[3] Bytes are not necessarily eight bits in length. Such a value is more strictly known as an *octet*. C only requires that all bytes contain *at least* 8 bits.

result in an increase in the number of bytes required to represent a given set of values.

Structure types take up as least as much space as the sum of their constituent types. They may take more because of the presence of wasted space, known as *padding*, which is added between and after members to ensure that alignment constraints are met.[4] An array type $\tau\texttt{[}n\texttt{]}$ takes size $n * |\tau|$. Pointer and function types do not have minimum sizes specified for them, except `(char *)` pointers must be able to store at least 32768 values.

In short, we require the existence of a family of functions, parameterised on types τ, that calculate values from representations. We will call these $V_\tau : \texttt{byte}^{|\tau|} \rightarrow \tau$. The various V_τ are surjective, but are not required to be injective, nor total, except for $V_{\texttt{unsigned char}}$, which is required to be a total bijection. In the reverse direction, there is no "representation function" *per se*, and the implementation is at liberty to choose from representations that give back the correct value. The existence of this state of affairs can not be hidden from the abstract semantics because we must allow values in memory to be taken apart and examined byte-by-byte.

Finally, the various integral types are required to be represented using a "binary numeration system", where signed types are further allowed to be two's complement, one's complement, or sign-magnitude. This requirement allows C programmers to fiddle with integral values at the bit level as well as by manipulating their values more abstractly at the level of arithmetic.

2.5 Expressions and their types

Having specified all the possible forms of the values that can arise and be manipulated in C programs, we now turn to discussing the language used in these programs to denote these values. For the moment, we ignore C's statements, because these do not denote values that the programmer is aware of. At a higher level of abstraction, statements *do* denote values, namely state transformers, but these are values only implicitly present in the abstract machine.

C's expressions can denote objects in the abstract machine's memory as well as values, so our typing judgements will not be simply that an expression is of a given type τ, but rather that it may yield a value of type τ, or that it may yield a reference to an object in memory of type τ. This latter

[4] Alignment, not modelled here, is defined as "a requirement that objects of a particular type be located on storage boundaries with addresses that are particular multiples of a byte address." [ISO90, §3.6]

we will write as $\mathbf{obj}[\tau]$. This is similar to the treatment given to lvalues in the Algol-like languages described in [Ten91]. There Tennent uses **var** for lvalues, but obj here is more in tune with the standard's use of the term "objects".[5]

Expressions may include variables, so our type judgements will include reference to variable-typing contexts, which we will write Γ, Γ_1 etc. Such contexts are finite maps from variable names to types. Further we will assume that these contexts map identifiers to non-`void` and well-formed types. Expressions may also contain references to the C program's declared functions, so another map is needed, this from function identifiers to types. These will be written Φ, Φ_1 etc. As before, we will also need structure information so that typing judgements will be of the form $\Gamma, \Phi, \Sigma \vdash e : \tau$ or $\Gamma, \Phi, \Sigma \vdash e : \mathbf{obj}[\tau]$.

As we introduce C's typing rules, we will simultaneously introduce C's expression syntax. We will use id to range over variable, function and `struct` member identifiers, and n over numbers.

The first rules are the simple base cases: numeric constants, the null pointer constant, and variables. (The standard provides for methods of specifying constants in unsigned, `long` and floating point types; we omit these for brevity's sake.)

$$\frac{}{\Gamma, \Phi, \Sigma \vdash n : \texttt{signed int}} \quad \frac{}{\Gamma, \Phi, \Sigma \vdash 0 : \tau *} \quad \frac{}{\Gamma, \Phi, \Sigma \vdash \texttt{Var}\,\mathsf{id} : \mathbf{obj}[\Gamma(\mathsf{id})]}$$

As explained earlier, we take the names of the functions defined in a program to be literals of function type. Because of the way in which we separate function references from pointers, we also need a distinct rule for null function references.

$$\frac{}{\Gamma, \Phi, \Sigma \vdash \texttt{Funref}\,\mathsf{id} : \Phi(\mathsf{id})} \quad \frac{}{\Gamma, \Phi, \Sigma \vdash 0 : \tau_1 \times \tau_2 \times \ldots \tau_n \to \tau}$$

Pointers in C can be generated by taking the address of an argument. For non-`void` types, this operation can in turn be reversed by dereferencing. Pointers to `void` are used as generic pointers in C, but before being able to follow such a pointer, it must be coerced back to an appropriate type.

$$\frac{\Gamma, \Phi, \Sigma \vdash e : \mathbf{obj}[\tau]}{\Gamma, \Phi, \Sigma \vdash \texttt{\&}e : \tau *} \quad \frac{\Gamma, \Phi, \Sigma \vdash e : \tau * \quad \tau \neq \texttt{void}}{\Gamma, \Phi, \Sigma \vdash * e : \mathbf{obj}[\tau]}$$

With the exception of arrays, if an expression yields a reference to an object in memory, then it can also yield a value of the same type. As previously

[5] Of course this has nothing to do with object orientation.

discussed, arrays are not first-class values in C, and it is not possible to manipulate them directly. Instead, references to array objects decay to yield a pointer to the first element of the array.

$$\frac{\Gamma, \Phi, \Sigma \vdash e : \textbf{obj}[\tau] \quad \tau \text{ not an array type}}{\Gamma, \Phi, \Sigma \vdash e : \tau} \qquad \frac{\Gamma, \Phi, \Sigma \vdash e : \textbf{obj}[\tau\,[n]]}{\Gamma, \Phi, \Sigma \vdash e : \tau *}$$

Member selection expressions are another illustration of the second-class nature of arrays. If the `struct` on the left hand side of the dot denotes an object, then there is no problem, thus:

$$\frac{\Gamma, \Phi, \Sigma \vdash e : \textbf{obj}[\texttt{struct}\ s] \quad (\mathsf{id}, \tau) \in \Sigma(s)}{\Gamma, \Phi, \Sigma \vdash e.\mathsf{id} : \textbf{obj}[\tau]}$$

However, if the `struct` is just a value, as can happen if a `struct` is returned by a function, then it is not possible to manipulate member arrays at all.

$$\frac{\Gamma, \Phi, \Sigma \vdash e : \texttt{struct}\ s \quad (\mathsf{id}, \tau) \in \Sigma(s) \quad \tau \text{ not an array type}}{\Gamma, \Phi, \Sigma \vdash e.\mathsf{id} : \tau}$$

Note that for non-array member types, there will be two ways of demonstrating that a member selection expression has a value type whenever the `struct` expression has an object type. This does not reflect any semantic ambiguity.

Before introducing the remainder of C's expression syntax, the (dynamic) notion of *coercion* must be described. The process of coercion occurs when a value in one type is coerced to become another value in a different type. Typically, the purpose of this is to preserve values, and in the integral types coercions from a smaller type to a superset will enjoy this property. In other circumstances, the result can not be the same value because the new type does not include the value in question. Depending on the types in question this can cause an error, or force the result to be somehow "appropriate". For example, a negative value in a signed type is coerced to the corresponding unsigned type by adding the largest possible unsigned value to the negative number. On the other hand, coercions between floating point and integral values admit the possibility of failure if the value being converted is too large or small, but otherwise try to return the integral value closest to the floating point one.

The expression syntax for coercion is the *type-cast*:

$$\frac{\Gamma, \Phi, \Sigma \vdash e : \tau_0 \quad (\tau_0 \text{ and } \tau \text{ both scalar}) \vee (\tau = \texttt{void})}{\Gamma, \Phi, \Sigma \vdash (\tau)e : \tau}$$

where the scalar types are the arithmetic types as well as pointers and function references.

Casting to `void` will not produce a value (unlike SML's `unit` type, the `void` type does not have any inhabitants), but allows one to indicate that one is not interested in the value of an expression, and presumably that one is evaluating it solely for its side effects.

An important static notion that accompanies that of coercion is that of implicit coercibility. This allows one to write assignment statements and function calls without having to ensure that the types agree exactly. We will denote this equivalence relation by the symbol $=_{\tau c}$. Omitting reflexivity, symmetry and transitivity, the rules for $=_{\tau c}$ are:

$$\frac{\tau_1 \text{ and } \tau_2 \text{ both arithmetic types}}{\tau_1 =_{\tau c} \tau_2} \qquad \frac{}{\texttt{void*} =_{\tau c} \tau\texttt{*}}$$

This allows us to type function calls:

$$\frac{\begin{array}{c}\Gamma, \Phi, \Sigma \vdash e : \tau_1 \times \tau_2 \times \ldots \tau_n \rightarrow \tau \\ \forall i.\ 1 \leq i \leq n \ \Rightarrow\ \exists \tau'.\ \Gamma, \Phi, \Sigma \vdash e_i : \tau' \ \wedge\ \tau_i =_{\tau c} \tau'\end{array}}{\Gamma, \Phi, \Sigma \vdash e(e_1, e_2, \ldots e_n) : \tau}$$

C has a wide variety of binary and unary operators, and one ternary operator, the conditional expression (e_1 ? e_2 : e_3). These are mainly concerned with operations on scalar types such as arithmetic, and there is little in their static semantics worth examining in depth. None of these operators produce or require object types as arguments. For example, the rules for the various binary operators all take the form

$$\frac{\Gamma, \Phi, \Sigma \vdash e_1 : \tau_1 \quad \Gamma, \Phi, \Sigma \vdash e_2 : \tau_2 \quad P_{\odot}(\tau_1, \tau_2, \tau)}{\Gamma, \Phi, \Sigma \vdash e_1 \odot e_2 : \tau}$$

where $P_{\odot}$ is the side condition on the types concerned for the operator in question.

The assignment and post-increment expressions are the last expression forms we will examine in any depth. C's simple assignment can be augmented with binary operators so that instead of `x = x + 4` one can write `x += 4`. Depending on the form of the expression on the left hand side (LHS) of the assignment, these *compound-assignments* can make a semantic difference, because of the possible presence of side effects. Simple assignment is typed as follows:

$$\frac{\Gamma, \Phi, \Sigma \vdash e_1 : \textbf{obj}[\tau] \quad \Gamma, \Phi, \Sigma \vdash e_2 : \tau_0 \quad \tau_0 =_{\tau c} \tau \quad \tau \text{ not an array type}}{\Gamma, \Phi, \Sigma \vdash e_1 \texttt{=} e_2 : \tau}$$

Compound assignment's rule is

$$\frac{\Gamma, \Phi, \Sigma \vdash e_1 : \textbf{obj}[\tau] \quad \Gamma, \Phi, \Sigma \vdash e_1 \odot e_2 : \tau_0 \quad \tau_0 =_{\tau c} \tau \quad \tau \text{ not an array type}}{\Gamma, \Phi, \Sigma \vdash e_1 \odot\!\texttt{=}\, e_2 : \tau}$$

The post-increment operation requires an object of scalar type to operate on, and does not change the object's type. Its rule is

$$\frac{\Gamma, \Phi, \Sigma \vdash e : \textbf{obj}[\tau]}{\Gamma, \Phi, \Sigma \vdash e\texttt{++} : \tau}$$

where τ is a scalar type but not a function reference. The post-decrement operator has the same rule.

Finally, four more expression forms are worth mentioning here. All are syntactic sugar. Array indexing `a[i]` is rewritten to `*(a + i)`,[6] and a form of member selection for use with pointers, `ptr->fld`, stands for `(*ptr).fld`. C also provides pre-increment and pre-decrement operators, written `++x` and `--x`. These are abbreviations of `x += 1` and `x -= 1` respectively.

2.6 Statements

C has a fairly minimal set of statements. Advocates of the language tend to take pride in its small set of keywords. Here we reduce the set still further by omitting description of the `switch` and `goto` statements. In both cases this decision was made because of the difficulties that formalising these forms would have posed. Nevertheless we feel that most of C's essence remains even with the omission. The forms we will consider are:

$$\begin{array}{rcl} s & ::= & \texttt{;} \mid e\texttt{;} \mid \texttt{while}\ (e)\ s \mid \texttt{for}\ ([e_1];\ [e_2];\ [e_3])\ s \mid \texttt{do}\ s\ \texttt{while}\ (e)\texttt{;} \\ & & \texttt{if}\ (e)\ s_1\ [\texttt{else}\ s_2] \mid \texttt{break;} \mid \texttt{continue;} \mid \texttt{return}[e]\texttt{;} \\ & & \{d_1 \ldots d_n\ s_1 \ldots s_n\} \end{array}$$

Here we use e, e_1 etc. to range over expressions, s, s_1 etc. to range over statements and d, d_1 etc. to range over declarations. The square brackets ([ ]) indicate an optional component. The statement ; is the empty statement. We will henceforth treat the `if`-statement as always having an `else` branch, understanding that the `else`-less form is simply an instance of the former with the empty statement substituted for s_2. The e; form is an

[6] As before, arithmetic on pointer values is thus built in at a very low level in C. The commutativity of addition also allows `array[i]` to be written as `i[array]`, a common trick if one's goal is obfuscation.

expression statement, whereby any expression can be included at the statement level. Any such expression has its value ignored, but its evaluation may still cause side effects.

The `for` loop begins by evaluating the first expression (if any). It then loops while the second expression is true, or indefinitely if the second expression is absent. Each time the body of the loop completes, the third expression (if any) is evaluated. A common idiom is thus

$$\texttt{for}\ (i = 0;\ i < limit;\ i\texttt{++})\ s$$

The `break` statement exits an enclosing loop, while `continue` causes the rest of an enclosing loop to be skipped, but the flow of control then returns to the top of the loop. Where an expression is absent in a `for` statement, we model this by inserting the number one where the expression is expected. Such an expression has no effect in the first and third positions, as required, but by being non-zero will cause indefinite execution in the second position, also as required.

There are just two constraints imposed on the expressions that appear in statements: the type of the guard expressions in the looping and conditional statements must be scalar, and the type of the expressions that appear in `return` statements must be implicitly coercible to the return type of the function in which they appear. All other expressions must have a type but are not otherwise constrained.

C's declarations come in two forms. They either declare a variable to have some type (optionally initialising it with some value of an implicitly coercible type), or they declare a new `struct` type. Their syntax is famously baroque and we omit any description of its complexities here. It is declarations that affect the contexts that are used to type expressions. Variables and `struct` tags both have static scope, meaning that one can always tell what entity is being referred to when a name occurs in a program text. C does not allow nested functions, so typing contexts will always be a combination of those entities that are declared locally (in the current function body), and those that are declared at the top level.

Finally, our model does not allow `static` declarations inside function bodies. Instead we require that these be modelled by declaring global variables (possibly renamed to avoid name clashes), and enforcing the restriction that they only be referred to in the functions where they are declared. This transformation is easily accomplished as a program is parsed.

2.7 Mechanisation of the static semantics

C's static semantics is not particularly complicated. By way of comparison, the parametric polymorphism and higher order functions present in SML make that language's static semantics much more complicated. Yet there is quite a long tradition of work on SML in HOL [Van96, Sym93, Van93]. For example, Van Inwegen's work on SML specifies both that language's static and dynamic semantics and then develops a proof of type soundness relating those two semantics. The same work involved the creation of many of the theorem-proving tools that our own work on C relies on, such as HOL's mechanism for defining mutually recursive types, and functions over those types.

More recently, work on the mechanisation of Java's semantics has been pursued by both Syme in his own DECLARE tool [Sym97b], and Nipkow and von Oheimb [NvO98] in Isabelle. This work involved the mechanisation of the complicated object-oriented static and dynamic semantics of a subset of Java, followed by the proof of this system's type soundness. C's static semantics is not such an inspiring example, and although we do later prove a type preservation property for C, this is not a particularly useful result in the light of the many places where the language's dynamic semantics is undefined.

In any case, such a result necessarily relies on the definition of the dynamic semantics. Before the dynamic semantics can even be approached, there is a substantial amount of mechanisation to be done first. The remainder of this section will describe this work, as it was done in the development of the Cholera mechanisation.

After defining the abstract data type corresponding to types (using the previously mentioned tools, described in [Gun93]), but omitting the floating point types, we define well-formedness for types using Harrison's inductive definitions package [Har95b]. At this point an interesting design choice presents itself. We could use the definition of this predicate over C types as the basis for the definition of a new type, that of well-formed types, and then drop the original definition of types entirely.

This is just the first place where the issue of sub-typing, and how one should approach it, raises its head. In general our approach has been to avoid defining new types such as these. HOL's type definition method requires their development as subsets of existing types, and once a type is actually defined, the link between the old and new type is tedious to exploit. Instead we define functions over larger types but ignore their values for elements of the domain not in the relevant subset. The end result is an avoidance of type proliferation (along with all of the accompanying ho-

momorphisms from type to type), but an acceptance that a great number of theorems will include preconditions on their variables to the effect that they satisfy some predicate.[7]

This decision also influences inductive definitions. When defining the expression typing relation, we must assert that the variable-typing and function-typing maps (Γ and Φ in the presentation above) have only well-formed types in their ranges. These assertions appear in a number of the typing relation's base cases.

We define the expression typing relation without coping with the polymorphic nature of 0.[8] Instead we assume that some preliminary phase of parsing has deduced the correct type for occurrences of this constant, and that we are instead presented with three different expression forms. We would otherwise have to annotate expressions with their expected types when we came to define the dynamic semantics. Once defined, the expression typing relation needs a number of theorems proved about it. We prove that the types it assigns to expressions are all well-formed, and that if it assigns any type to an expression, then it will only assign one type. These are the most important theorems, but there are also a number of minor results to be proved.

One such theorem is a series of equations for the relation over the various possible expression forms. This is the automatic output of a specially written ML routine that does the same for any inductive relation. Given such a relation, along with the number of the argument where the type of interest is a parameter, and theorems about the type's constructors specifying that they are all injective and disjoint, the code generates an equational rewrite for each possible form in the type. In the example of expression typing, we naturally specify that the expression is the interesting parameter, because we typically want to use the relation to find an expression's type.

Indeed, the typing relation is one we could have defined directly as a primitive recursive function over the type of expression syntax. This would have been a function to `:bool`, not to `:CType` (the type of C types), but would have given us the same equations as the previously described code. On the other hand, this approach would not have given us the induction principle that the inductive definition provides, and this would have had to have been derived separately instead.

---

[7] The PVS theorem prover [ORS92] takes another course again; it allows the construction of new sub-types using predicates to select subsets of larger types. This then introduces the problem of requiring the discharge of type correctness conditions.

[8] Recall that zero can be either a signed integer, a null pointer or a null function reference.

If a relation can not be defined as a primitive recursive function over one of its parameters, one must use an inductive definition. In this case, the ML code for generation of equational theorems may well produce equations that can not be used as rewrites. Naturally, this happens with the definition of the dynamic semantics, where the rewrite equations generated from the definition of the dynamics include one which loops on `while` statements.

Finally, the mechanisation of C's statics must also specify the nature of values in the abstract machine. Here the standard's deliberate under-specification of details is naturally met by HOL's analogous facilities for definition of constants. In particular, the `new_specification` function allows one to define a constant in a type such that the only thing known about the new value is that it satisfies some predicate. The user defining the value must prove that such a value exists, but subsequently will only be able to rely on the constant satisfying the predicate given.

Thus the defining theorem for the constant `CHAR_BIT`, which specifies the number of bits in a byte, is simply

$$\vdash \texttt{CHAR_BIT} \geq 8$$

In exactly the same way, C's various under-specified limits on type ranges can be expressed naturally and correctly in HOL.

The type size relation (`sizeof`) is also defined at this stage. Unfortunately the mechanisation of this does not allow for the possibility of padding, and simply sums the size of `struct`s' members. This inaccuracy in the mechanisation is due to the fact that the standard does not limit padding in any way (the standard line in situations such as this is that the issue is one of "quality of implementation") and the alternative would be to perform proofs where one carried around assumptions to the effect that padding amounts in `struct` types were not unreasonably large. We then define offsets for `struct` members, and prove the fact that all non-void, well-formed types have a size.

At this stage, our mechanisation introduces its greatest inaccuracy: the assumption that all of the various V_τ functions are injective. This is thus an assumption that there is a function in the reverse direction, from values to representations. This flaw in the mechanisation makes our semantics more deterministic than it should be. In the absence of this assumption, even expressions such as `x = 1` are potentially non-deterministic because of the possibility that the value chosen to represent one will differ in its unused bits from invocation to invocation. This difference can be "seen" by programs because of the ability to take memory apart byte by byte.

For all that this assumption may seem very strong, it is not unreasonable in the context of many C implementations. In particular, twos complement

for signed integers, which is now the norm for most implementations, gives an isomorphism between integers and bit-patterns, meaning that the used bits of an integral representation will always be uniquely determined. It is also quite uncommon for the basic integral types to have unused bits, meaning that implementations have little scope in which to vary their valuation functions. By omitting floating point values and `struct` padding, our mechanisation also removes other areas in which implementations may have scope to introduce such variation.

The advantage of making this assumption is not in the specification of the semantics, which as chapter three will demonstrate is not unduly dependent on this aspect of the abstract machine, but is rather that it makes it easier to discuss determinism at the higher levels of expression evaluation. In particular, this assumption allows us to avoid qualifying all of our later assertions with preconditions about valuation functions.

While our mechanisation does simplify some details, it still accurately reflects a great deal of the under-specification called for by the standard. This means in turn that it is a useful tool for reasoning about a wide range of programs on a wide range of possible implementations. In particular, we have not simplified the language to the point where our model is equivalent to just one possible implementation.

Chapter 3

Dynamics

We describe C's dynamic semantics, starting with a full description of the components that go to make up a program state, continuing with a discussion of undefinedness, and then giving the semantics of expressions, statements and declarations. Again we conclude by describing the mechanisation of the semantics in the theorem prover.

3.1 The program state

Ultimately, a semantics gives a program meaning by describing either the way in which it transforms a "program state", or the way in which it interacts with its environment, or both. As our model doesn't formalise the C language's library, system calls to achieve I/O are omitted and we consider only transformations of program states as the basis for our semantics. These states model the state of an abstract computer on which the described program is being run. In its simplest form, a model of program state might include only a description of the abstract computer's memory, but here we describe why we find it necessary to include other components as well.

If a formal semantics is to retain the connection between the high level syntax of the language and what happens to memory, then it must also include the sort of details that one might at first associate with compiler symbol tables. The types of variables, the members that go to make up `struct` types, and the mapping from variable names to locations in memory must all be recorded. These details make up what is commonly known as the *environment*. However, in what follows, we shall loosely use the term "state" to refer to both the environment and the contents of memory. After all, both can be seen to change as a program executes.

3.1.1 Accessing memory

Our model of memory is a map from addresses to byte values. We know further that addresses within an object support a limited set of arithmetic operations, so it is natural to model addresses as numbers. Our model must support a value for the null pointer constant, and we choose 0 to fill this rôle. This does not mean that the bit-by-bit representation of the null pointer is required to be all zeroes. Rather, the ranges of our $V_{\tau*}$ valuation functions are the natural numbers. This imposes no requirements on the representation of pointers as they appear to the programmer.

It is easy for a C program to access memory in ways that are dangerous. For example, one can apply the address-of operation (`&`) to local variables, generating a pointer value that will continue to point at memory after the containing block has finished. It is illegal to manipulate such a pointer after the memory it points is no longer "live". Further, it is also illegal to refer to memory that has not been initialised, even if that memory is properly in scope. Therefore, memory must support being in one of three possible states: inactive, active but uninitialised, or value-holding.

One can not do anything with inactive memory. Even pointers to it are not valid and it is illegal to manipulate such values. This may seem an unduly restrictive constraint, but it arises to accommodate implementations on segmented architectures where invalid address values can actually cause processor exceptions (as happens on the Intel '286 chip, for example). Naturally, though one can not read uninitialised memory, one can write to it. This initialises it, making it available for further reading. An issue of some contention among standard committee members (as witnessed on the Usenet group `comp.std.c`) is whether or not this should be changed for the new version of the standard to allow lvalues of type `unsigned char` to access uninitialised memory. Such a move has some sense to it because we already know that all possible bit patterns have a possible value in the `unsigned char` type. Therefore, reading random values into such an lvalue from an implementation's stack would not cause processor exceptions.

In either case, given that the default `char` type, and hence that of the characters in string literals, is possibly `signed char`, the program in figure 3.1 is possibly undefined, and certainly not strictly conforming. Given that the program in question represents quite a natural idiom, it seems unreasonable to insist that it really have such an unclear status. Strict conformance would only be assured by explicitly initialising all of the array before using it. This seems a failing in the current standard, and it is not clear that the issue will be entirely resolved with the next version to be published. In this presentation we hew to the official line, and state that all

reads of uninitialised memory result in undefined behaviour.

```
#include <string.h>

int main(void)
{
  /* declare s1 and s2 to be of string_carrier type */
  struct string_carrier {
    char the_string[30];
  } s1, s2;

  /* copy "Hello world" string into part of s1's string array */
  strcpy(s1.the_string, "Hello world");

  /* assign s1 into s2, and invoke undefined behaviour because
     not all of s1's the_string array has been initialised, and
     assignment must read all of the array's values. */
  s2 = s1;
}
```

Figure 3.1: A program of dubious definedness

3.1.2 Side effects

In C, changes to memory come about solely through the action of side effects. These are created as the result of evaluating certain expression forms, principally assignment expressions. However, side effects are not applied immediately upon their creation, but can be kept pending. Nor do multiple side effects have to be applied in order.[1] The motivation for this is that implementations should have licence to make changes to memory in ways that are convenient for them. This licence may allow useful optimisations to be performed, for example.

We model this state of affairs by keeping a bag (or multi-set) of pending side effects as part of the state. As the side effects are generated, they are put into the bag. The bag is emptied in a non-deterministic order, and at non-deterministic times, subject only to the constraint that it must be empty when what is known as a *sequence point* is encountered. Sequence

[1]"Except as ... specified later ... the order in which side effects take place ... [is] unspecified."[ISO90, §6.3]

points occur in certain syntactically well-defined places, and will be flagged in the description that follows.

This non-determinism suggests a chaotic picture. However, the language definition imposes severe constraints on the way in which expressions can evaluate. The principal constraint is that "between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be accessed only to determine the value to be stored." [ISO90, §6.3] Violation of this constraint results in undefinedness.

It is worth noting that this is a constraint on the dynamic behaviour of the program. Though an expression such as `v + v++` will necessarily be undefined because it both refers to and updates the object denoted by `v`, it is not clear whether or not this is true of `*p + (i = 1)`, say, as it is impossible in general to determine whether or not `*p` will refer to `i`. Finally, note that the second sentence quoted above allows references to take place if they occur on the right-hand side of an assignment expression and the references are to the object being modified by the assignment. For example, this clause allows `i = i + 1`.

A formal semantics of C must model this constraint as well as the more obvious rules given earlier. To do this, we keep track of three state components:

- the pending side effects
- those parts of memory which have been updated (the "update map")
- those parts of memory which have been referred to (the "reference map")

The pending side effects component is a bag, as the same side effect might occur twice in a given evaluation, though such a situation will eventually result in undefinedness when the effects are applied. The update map is a set of addresses, as no evaluation will be allowed to update the same location twice. The reference map is another bag, as multiple references to the same location can legitimately occur. As we shall see, we need to know how many references were made to a particular location, not just whether or not something has been referred to.

There are four different ways in which these components can change in the evaluation of an expression:

- When a non-array lvalue becomes a value, the reference map is increased to reflect the reference of the object denoted by the lvalue.[2] If the part of memory referred to is in the update map, this causes undefinedness.

- When a side effect is applied, it is removed from the pending side effects bag, and the update map is increased, recording the fact that part of memory has just been changed. If that part of memory has already been updated or referred to, this causes undefinedness.

- When an assignment completes its evaluation, a side effect to update the appropriate part of memory with a new value is added to the bag of pending side effects. Assignment expressions keep track of references made on their right hand sides, and those that were to the object to be updated are removed from the reference map. Failure to do this would cause the side effect created as a result of evaluating `i = i + 1` to clash with the reference to `i` on the expression's RHS.

  Because the reference map records a count of the number of times a piece of memory has been referred to, this deletion of references may still leave references recorded. Using only a set for the reference map would allow

  ```
  i + (i = i + 1)
  ```

  to avoid revealing its undefined nature. A possible evaluation would have the `i` on the assignment's RHS remove the record of a previous reference to `i` on the LHS of the addition.

- When the bag of pending side effects is empty, and a sequence point is reached in an expression's syntax, the reference map and update map are "zero-ed", thereby allowing a new sequence of reference and updates in the next phase of execution. If a sequence point is reached, and the bag of pending side effects is not empty, it will need to be emptied before the next stage of the expression can be evaluated.

3.1.3 Notation—general observations

Each section of the semantics defines its own operators and miscellaneous notation, but there are a number of general observations about notation that can be made here. Each section of the semantics presents an "arrow"

[2] Array lvalues become pointers to their first element; this transformation does not require a reference to memory.

relation of the form $\langle v_0, \sigma_0 \rangle \rightarrow \langle v, \sigma \rangle$, where σ_0 and σ are the initial and final states, and where v_0 is the form of C syntax being defined. The nature of v depends on the precise relation being defined.

The following tables summarise the functions used in the rule definitions. Table 3.1 lists “state functions”, which calculate commonly needed values from the current state of the program. We have already encountered structure information, variable, and function constant typing maps. We write the components of specific states by using the appropriate Greek letter, subscripted by the state to which it belongs. Thus, Γ_σ represents the variable typing information for state σ. We will use the notation $\sigma[C := E]$ to represent the state that is identical to σ except that its component C has been replaced with a new value E.

Further note that in table 3.1, components marked with an asterisk are those which have global variants present in the state. These do not change in the course of a program’s execution, but form the basis for the local contexts, which do change as the flow of control enters and exits statement blocks and function bodies. Such global components are marked with a ^{g} superscript when they appear.

We will often have cause to refer to map “slices”. Memory, for example, is often accessed “bytes at a time”. We will refer to such a slice by writing $M\langle x \ldots y \rangle$ to denote the sequence of values between indices x and y inclusive. We will also use an update notation for slices thus: $M[\langle x \ldots \rangle := m]$. This denotes the same map as M, except that it has been overlaid with the value sequence m, starting at address x.

A	variable addresses*	Γ	variable typing*
I	initialised addresses	Λ	allocated addresses
M	memory map	Σ	structure information*
Π	pending side effects	R	the reference map
Υ	the update map	Φ	function information

Table 3.1: state components

Because of the presence of bag components R and Π in program states, we often have cause to write expressions denoting bags and operations on them. We use set notation to express bag values where each element is present just once, so that $\emptyset$ will stand for the empty bag. When in a context expecting sets (such as an argument to intersection), we implicitly coerce bags back to sets. The resulting set includes an element if that element occurs one or more times in the bag. Where we have a bag of numbers,

we also allow the slice notation, so that $B\langle x \dots y\rangle$ denotes the sub-bag of B which contains only those elements of B which are in the range $x \dots y$. We write $x \in B$, with B a bag, to mean that x occurs at least once in B. We use $+$ and $-$ to denote the operations of bag addition and subtraction. (When bags are seen as functions with range $\mathbb{N}$, these operations are simply seen as point-wise applications of the corresponding arithmetic operations. Recall also that in $\mathbb{N}$, $n - m = 0$, for all $m \geq n$, so that $b_1 - b_2 = b_1$ if b_1 and b_2 are disjoint.)

Finally, we will define a few "helper" functions in the course of the chapter. These are typically used as abbreviations for relatively complicated expressions on states, but are not particularly interesting in their own right. These are summarised in table 3.2.

`alloc`	returns the address of some unallocated space
`decl_var`	declares a variable, allocating space and type information
`inst_parms`	installs parameters in a function call
`offset`	calculates `struct` member offsets
`okVal`	checks a value is "acceptable" in its type
`OpSem`	implements binary operator semantics
`UnOpSem`	implements unary operator semantics

Table 3.2: state "helper" functions

3.2 Undefinedness

When the standard's definition of C states that a behaviour is undefined, it is restricting the language. A program that invokes undefined behaviour is not really a C program at all. However, as we have already discussed in chapter 2, we can not use a static semantics to detect whether or not a program is undefined. Instead, our dynamic semantics must deal with the possibility that the syntax it is providing a meaning for is actually undefined. Here we will examine how this might be done, and explain the approach eventually chosen. We will illustrate each option with rules for division in a hypothetical language defined using a big-step semantics. We assume that this language leaves the result of division by zero undefined, as does C.

The simplest, but also most obviously invalid, approach is to ignore the

possibility of undefinedness entirely. Here we would write

$$\frac{\Gamma \vdash e_1 \Rightarrow v_1 \quad \Gamma \vdash e_2 \Rightarrow v_2}{\Gamma \vdash e_1 \div e_2 \Rightarrow v_1/v_2}$$

We assume that our underlying logic gives division by zero some acceptable meaning. The problem with this approach is that we have now defined a semantics for a larger language than expected. This might be acceptable if we are only interested in defining the behaviour of programs that we know to be correct, but is obviously inappropriate if we want to use the semantics to tell us which programs are defined.

A second approach would be to add a side condition forbidding the undefined circumstance. This would result in a rule such as

$$\frac{\Gamma \vdash e_1 \Rightarrow v_1 \quad \Gamma \vdash e_2 \Rightarrow v_2 \quad v_2 \neq 0}{\Gamma \vdash e_1 \div e_2 \Rightarrow v_1/v_2}$$

This is an adequate solution in a language without non-determinism because it allows us to detect invalid programs by determining whether or not they are assigned values by the evaluation relation. If they are not, then it will be due to a rule such as the one above. However, in a language with non-determinism, such a rule will likely give misleading results. Let us extend our example with program states, and allow the possibility of side effects. We might then have the following rules:

$$\frac{\Gamma \vdash e_1, \sigma_0 \Rightarrow v_1, \sigma_1 \quad \Gamma \vdash e_2, \sigma_1 \Rightarrow v_2, \sigma \quad v_2 \neq 0}{\Gamma \vdash e_1 \div e_2, \sigma_0 \Rightarrow v_1/v_2, \sigma}$$

$$\frac{\Gamma \vdash e_2, \sigma_0 \Rightarrow v_2, \sigma_1 \quad \Gamma \vdash e_1, \sigma_1 \Rightarrow v_1, \sigma \quad v_2 \neq 0}{\Gamma \vdash e_1 \div e_2, \sigma_0 \Rightarrow v_1/v_2, \sigma}$$

These allow for evaluation of division expressions in one of two orders, and it is possible that these different evaluation orders might result in different values for v_1 and v_2. But now consider the situation where one order results in v_2 having value 0. Our semantics for our language will discard the possible evaluation which is undefined, and return only the evaluation which terminates correctly. It seems unlikely that any language specification should require non-determinism to "dodge" undefinedness in this way, and it is certainly the case that C does not.

It is clear that to model undefinedness accurately, we will need to model its occurrence explicitly. Checking that a program is well-behaved then consists of confirming that undefinedness is not a possible result of evaluation. In a semantics such as our example, this modelling of undefinedness is best

achieved by allowing some special undefined value to appear on the right hand side of the evaluation arrow. In a small-step semantics, it is possible to avoid this. Instead one uses the approach above where transitions can not occur because of side conditions. An evaluation is defined only if it reduces to a value (rather than stopping short at some intermediate stage). Although our expression semantics is small-step, the statement semantics we present is big-step, and this means that we do use an explicit undefinedness value. This is written $\mathcal{U}$.

Finally, we must determine the status of a program which is undefined in some paths of computation, and not others. In fact, C requires that if any of the possible evaluations leads to undefinedness, then the program must be considered undefined. This follows from the definition of undefined and unspecified behaviours given in [ISO90]. Unfortunately this is impossible to express within an inductive definition, so the following definition of the language's dynamic semantics is only implicitly overlaid with this requirement.

3.3 Expression evaluation

Expression evaluation uses a small-step semantics. The relevant relations are $\to_e$, and its transitive and reflexive closure, $\to_e^*$. The action of $\to_e$ can be seen as a gradual transformation of a piece of syntax into a value. For this relation to be properly typed, values must themselves be considered as part of the expression abstract syntax type.

We have already seen that a value is a sequence of bytes. In the context of the dynamic semantics, we shall extend this notion somewhat so that values are accompanied by their types. Thus, values that appear in the expression syntax will be written as $\underline{(m, \tau)}$, where m is a list of bytes, and τ is the type of the value. The underlining is used to emphasise that this is the final result of an evaluation. The accompanying type is necessary so that the overloaded operators know how they should be applied (adding one to a pointer, for example, has quite a different behaviour to adding one to an integer). The alternative would be to extend expressions so that the dynamic semantics was presented with expressions already tagged with the types assigned to them by the static semantics. Given that values are to inhabit our syntax trees as they are evaluated, this would result in values still having types associated with them, just as every other component of the tree would. This alternative method of presentation has its appeal (it would cope well with polymorphic zero), but was not pursued due its dependency on the static semantics.

Values are not the only addition we need to make to the abstract syntax of expressions. Just as we extended our judgements in the static semantics to handle lvalue types (the $\textbf{obj}[\tau]$), we extend our expression syntax to handle values that correspond to lvalues, or references to objects. Our new `LV` constructor will take an address and a type as arguments, thus uniquely specifying an object in memory. If we have $\Gamma_\sigma, \Phi_\sigma, \Sigma_\sigma \vdash e : \textbf{obj}[\tau]$, then we expect $\langle e, \sigma \rangle \to_e^* \langle \texttt{LV}(a, \tau), \sigma' \rangle$, for some a and σ'.

Another additional form is the "(r)value required" constructor `RVR`, which takes a single expression as an argument. This is a trick of the formalisation, principally used at the top level to ensure that expressions do evaluate to values, and don't stop short as just lvalues. For reasons that should become clear, it is not possible to have a general reduction that allows $\langle \texttt{LV}(a, \tau), \sigma \rangle \to_e \langle v, \sigma' \rangle$, where v is the value of the object denoted by the lvalue. If this rule were part of the definition of $\to_e$, it could occur anywhere in an expression's syntax tree. This would lead to blocking in assignment expressions, among other places, as an lvalue is required on the LHS of an assignment expression if it is to reduce. In the presence of the putative rule above, an lvalue could reduce to a value regardless of its situation in the wider context.

3.3.1 Expression evaluation contexts

Following the example of other presentations of reduction semantics (e.g., [Gor95], and ultimately [FF86]) our expression semantics makes use of evaluation contexts. Informally, an evaluation context is a piece of syntax "with a hole in it". Contexts provide a convenient way to generalise a whole family of evaluation rules where independent reductions occur in sub-expressions. The following rule, using the $\mathcal{E}$ context, captures most of the ways in which evaluation can proceed in the sub-terms of an overall expression:

$$\frac{\langle e_0, \sigma_0 \rangle \to_e \langle e, \sigma \rangle}{\langle \mathcal{E}[e_0], \sigma_0 \rangle \to_e \langle \mathcal{E}[e], \sigma \rangle}$$

This rule should be read as "If expression e_0 can reduce to e, altering state σ_0 to σ in the process, then the expression formed by inserting e_0 into the hole of the context $\mathcal{E}$ can start in σ_0 and reduce to the corresponding expression with e in place of e_0, finishing in state σ." The allowed forms for $\mathcal{E}$ are:

$$\begin{array}{rl}\mathcal{E}[-] ::= & - \odot e \mid e \odot - \mid \Box - \mid - \texttt{\&\&}\, e \mid - \texttt{||}\, e \mid - \texttt{,}\, e \mid \\ & -\texttt{++} \mid -\texttt{--} \mid - \texttt{.}\, \mathsf{id} \mid - \odot\texttt{=}\, e \mid - \texttt{=}\, e \mid (\tau)- \mid \texttt{RVR}- \mid \\ & - \texttt{?}\, e_1 \texttt{:}\, e_2 \mid -(e_1, \ldots, e_n) \mid e(e_1, \ldots, -, \ldots e_n)\end{array}$$

Here $\odot$ stands for any of the standard binary operators (+, *, | etc.), and $\Box$ stands for any of the unary operators -, !, ~, &, *.

The sequenced behaviour of the logical operators && and ||, as well as the comma operator and the ?: operator is apparent here; only an evaluation that looks at the first argument is acceptable. Conversely, there are two contexts for the binary operators, giving rise to a significant non-determinism; the evaluation of the operands can be interleaved at all levels. In particular, evaluation of $(a + b) + (c + d)$ can see evaluation of the sub-expressions a, b, c and d occur in *any* order.

The $\mathcal{E}$ context is also used in the propagation of the $\mathcal{U}$ value.

$$\frac{}{\langle \mathcal{E}[\mathcal{U}], \sigma\rangle \rightarrow_e \langle \mathcal{U}, \sigma\rangle}$$

The $\mathcal{E}$ context doesn't extend as far as the right hand sides of (compound) assignment statements (the reason for this is explained in section 3.3.4), so we need to also add this as a location in which undefinedness is propagated upwards. (The rule for the compound assignment case is the same.)

$$\frac{}{\langle e_1 \texttt{=} \mathcal{U}, \sigma\rangle \rightarrow_e \langle \mathcal{U}, \sigma\rangle}$$

3.3.2 Base cases—values out of memory

In this section we present the expression evaluation relation's fundamental, "base case" rules. These are the rules which bring values out of memory and into syntax trees as they evaluate. Later sections define the ways in which values can be combined, and subsequently put back into memory.

Constants We assume that constants' types have already been statically determined. (With the exception of the polymorphic 0, this determination is entirely lexical.) We use the byte representation to value function to require that the value returned is a valid representation for the value.

$$\frac{V_\tau(m) = c}{\langle (c, \tau), \sigma\rangle \rightarrow_e \langle \underline{(m, \tau)}, \sigma\rangle}$$

As before, the underlining indicates that the syntax on the right-hand side is actually a value. This rule handles function reference constants (including the null reference) as well as numeric constants and the null pointer.

Variables and lvalues A variable denotes a piece of memory and its contents. We translate all variables into lvalues, using the following rule (the LV lvalue-constructor takes the variable's address and type as arguments):

$$\frac{}{\langle \mathsf{id}, \sigma \rangle \rightarrow_e \langle \mathtt{LV}(A_\sigma(\mathsf{id}), \Gamma_\sigma(\mathsf{id})), \sigma \rangle}$$

In many contexts, the semantics requires lvalues to become normal values. This is controlled by another relation, $\rightsquigarrow$. As before, this relation can not be part of $\rightarrow_e$ because there are contexts in which we don't want to have the lvalue information disappearing (assignment, for one).

$$\frac{\Upsilon_\sigma \cap r = \emptyset \quad r \subseteq I_\sigma \quad \mathtt{okVal}(\sigma, m, \tau) \quad \tau \text{ not an array type}}{\langle \mathtt{LV}(a, \tau), \sigma \rangle \rightsquigarrow \langle \underline{(m, \tau)}, \sigma[R := R + r] \rangle}$$

where $\quad r = \{a \ldots a + |\tau| - 1\}$
$\quad m = M_\sigma \langle a \ldots a + |\tau| - 1 \rangle$

The first three conditions above the line of this rule require that the lvalue denote an object which has not been updated in this phase of execution, that the part of memory being read is initialised, and that the bytes read out of memory constitute a valid value for the given type. We do not check that $s \subseteq \Lambda_\sigma$, as we claim that the initialisation map is always a subset of the allocation map. The definition of okVal, our check for "value validity" is:[3]

$$\mathtt{okVal}(\sigma, m, \tau) = \begin{cases} \exists a.\ V_\tau(m) = a \wedge (a \in \Lambda_\sigma \vee a - 1 \in \Lambda_\sigma) & \tau \text{ a pointer type} \\ \exists v.\ V_\tau(m) = v & \text{otherwise} \end{cases}$$

The definition of okVal accommodates the fact that pointers are allowed to point one past the end of allocated objects, even though dereferencing such values is illegal.

Our rule for $\rightsquigarrow$ is one of two where bit-sequences from memory are interpreted as values (the other is the rule for post-increment). In other rules, the valuation functions in this semantics can be assumed to be total because the values passed as arguments will have been generated by the semantics, ensuring their validity. In other words, it is our claim that the presence of a value $\underline{(m, \tau)}$ in the rules implies that V_τ has a value for m.

It is possible for the lvalue's address a to be zero. This corresponds to the null address, and will never be part of initialised memory (I_σ), thereby ensuring that this rule will not be satisfied. Instead, if any of the first three conditions in the above rule for $\rightsquigarrow$ do not hold then undefinedness results:

[3] Recall that V_τ is only necessarily total for $\tau = $ `unsigned char`, so that expressions such as $\exists n.\ V_\tau(m) = n$ are not tautologies.

$$\frac{(\Upsilon_\sigma \cap r \neq \emptyset \vee r \not\subseteq I_\sigma \vee \neg\texttt{okVal}(\sigma, m, \tau)) \quad \tau \text{ not an array type}}{\langle \texttt{LV}(a, \tau), \sigma \rangle \rightsquigarrow \langle \mathcal{U}, \sigma \rangle}$$

where $r = \{a \dots a + |\tau| - 1\}$
$m = M_\sigma\langle a \dots a + |\tau| - 1\rangle$

The case when the lvalue is of an array type needs to be dealt with separately. As our presentation of the static semantics should have already made clear, array lvalues are converted into pointers to their first element, which will necessarily share the same address.

$$\frac{V_{\tau*}(m) = a}{\langle \texttt{LV}(a, \tau\texttt{[}n\texttt{]}), \sigma \rangle \rightsquigarrow \langle \underline{(m, \tau*)}, \sigma \rangle}$$

The only issue to resolve is when the $\rightsquigarrow$ relation should be allowed to "fire". This is done with another evaluation context, called $\mathcal{L}$. $\mathcal{L}$ is the same as $\mathcal{E}$ except that it omits both arguments of assignment expressions[4], the arguments of ++ and --, the unary ("address-of") & operator, and the field selection . operator. The rule in $\rightarrow_e$ is

$$\frac{\langle e_0, \sigma_0 \rangle \rightsquigarrow \langle e, \sigma \rangle}{\langle \mathcal{L}[e_0], \sigma_0 \rangle \rightarrow_e \langle \mathcal{L}[e], \sigma \rangle}$$

Note that this rule means that lvalues at the very top level of an expression evaluation will not convert into normal values. This is handled specially at the statement level by wrapping expressions in the `RVR` constructor.

Structure values The field selection operator . pulls values out of memory through `struct` values. This operator is unique in that it can produce either lvalues or normal values depending on the nature of its first argument. When the operand is an lvalue, the rule is

$$\frac{(\mathsf{id}, \tau) \in \Sigma_\sigma(s)}{\langle \texttt{LV}(n, \texttt{struct } s)\ \texttt{.}\ \mathsf{id}, \sigma \rangle \rightarrow_e \langle \texttt{LV}(n + \texttt{offset}(\sigma, s, \mathsf{id}), \tau) \rangle}$$

where `offset` returns the offset of members within a `struct` type.

If the first argument to field selection is a normal value, then the result is also a normal value, created by copying out the list of bytes within the value corresponding to the field in question.

$$\frac{(\mathsf{id}, \tau) \in \Sigma_\sigma(s)}{\langle \underline{(m, \texttt{struct } s)}\ \texttt{.}\ \mathsf{id}, \sigma \rangle \rightarrow_e \langle \underline{(m', \tau)}, \sigma \rangle}$$

[4] While it should be clear why the left-hand side of the assignment should be immune to decay, it is less clear why the right-hand side needs this protection; this issue is addressed in section 3.3.4.

where $m' = m\langle o \ldots o + |\tau| - 1\rangle$
$o = \texttt{offset}(\sigma, s, \mathsf{id})$

Pointers As we have already seen, pointers are fundamental to the way in which lvalues are manipulated. There are two pointer specific operators: dereferencing (*) and taking addresses (&). The first of these takes a pointer value and returns an lvalue, subject to the constraints that the pointer not be a void pointer (of type (void *)). Such a pointer is used as the generic pointer, and can be used to store pointer values of any type, but it can never be dereferenced. Though this precondition is unnecessary given the assumption that the static semantics has already rejected badly typed programs, we include it here as a reminder of the typing rules.

$$\frac{\tau \neq \texttt{void}}{\langle \texttt{*}\underline{(m, \tau\texttt{*})}, \sigma\rangle \rightarrow_e \langle \texttt{LV}(V_{\tau\texttt{*}}(m), \tau), \sigma\rangle}$$

The & operator is the inverse of this, requiring an lvalue as an argument, and returning a pointer value.

$$\frac{V_{\tau\texttt{*}}(m) = a}{\langle \texttt{\&}(\texttt{LV}(a, \tau)), \sigma\rangle \rightarrow_e \langle \underline{(m, \tau\texttt{*})}, \sigma\rangle}$$

3.3.3 Value producing operators

The term "value producing operator" is meant to contrast with the side effect operators detailed in Section 3.3.4. Value producing operators generate new values, but do not modify the memory part of the program state. The first rule is the most general:

$$\frac{\odot \in B}{\langle \underline{(m_1, \tau_1)} \odot \underline{(m_2, \tau_2)}, \sigma\rangle \rightarrow_e \langle \texttt{OpSem}(\sigma, \odot, \underline{(m_1, \tau_1)}, \underline{(m_2, \tau_2)}), \sigma\rangle}$$

The operator function OpSem calculates the effect of the given operator ($\odot$ above), returning both the value and the type of the result. It will return $\mathcal{U}$ if the operation is undefined (as in division by zero, for example). It takes a state as its first argument because the addition, subtraction and relational operators may be applied to pointers. In this case the definedness of the operation is dependent not only on the pointer values in question but also on the state of memory. Further, the types of the operator's arguments are also required, so that the operator can perform appropriate conversions.[5] The

[5]For example, adding an int and a long will cause the int to be converted to a long and the result will also be a long

set B includes all of the standard binary operators (addition, subtraction etc.), but excludes `&&`, `||` and the `,` operator.

There is a very similar rule for the unary operators `~`, `!` and `-` (though here we do not need to pass states to the semantic function `UnOpSem` because none of these operators can yield pointer values):

$$\frac{\Box \in \{\texttt{\~{}}, \texttt{!}, \texttt{-}\}}{\langle \Box\underline{(m,\tau)}, \sigma\rangle \rightarrow_e \langle \texttt{UnOpSem}(\Box, \underline{(m,\tau)}), \sigma\rangle}$$

There is one further unary operator, the type-cast. This takes values and converts them to values of the specified type. This conversion is not possible for all pairs of types, but where it is possible, the semantics is very easy to specify in the abstract. C is the conversion function required by the semantics. As already discussed in chapter two, this will usually perform "useful conversion", even when the value to be converted is not in the destination type.

$$\frac{C : \tau_0 \rightarrow \tau \quad C(V_{\tau_0}(m_0)) = V_\tau(m)}{\langle \texttt{(}\tau\texttt{)}\,\underline{(m_0,\tau_0)}, \sigma\rangle \rightarrow_e \langle \underline{(m,\tau)}, \sigma\rangle}$$

Note that it is also possible for our conversion function C to return $\mathcal{U}$ when a value is not representable in a given type. (This can happen when attempting to convert `float` values to an integral type).

The rule for the comma-operator is the first to involve a sequence point. Before evaluation can proceed to the second argument of the expression, it must be the case that all pending side effects have been applied. Furthermore, when the evaluation then proceeds, the state must be updated to forget the references and updates made up to this point. This is because the restrictions on references and updates to the same object only hold between two consecutive sequence points.

$$\frac{\Pi_\sigma = \emptyset}{\langle \underline{(m,\tau)} \texttt{ , } e, \sigma\rangle \rightarrow_e \langle \texttt{RVR}(e), \sigma[R := \emptyset, \Upsilon := \emptyset]\rangle}$$

The `RVR` constructor forces an expression to become a normal value. It is used here to prevent the result of the comma-operator expression from being an lvalue. The only rule for `RVR` is

$$\frac{}{\langle \texttt{RVR}\underline{(m,\tau)}, \sigma\rangle \rightarrow_e \langle \underline{(m,\tau)}, \sigma\rangle}$$

If `RVR` is wrapped around an expression that evaluates to an lvalue, then the lvalue to value reduction, $\leadsto$, will be able to act upon it, because `RVR` is part of the lvalue context $\mathcal{L}$.

The logical operators: `&&` and `||` The `&&` and `||` operators implement the operations of "logical and" and "logical or" respectively. Each will "short circuit" if the first argument determines the value of the whole expression. Of course, the non-determinism available in other binary operators is not available here, and there is also a sequence point before the evaluation of the second argument if any. The evaluation of the first argument proceeds under the $\mathcal{E}$ and $\mathcal{L}$ contexts until it yields a value. The following rules specify what happens next. First the short circuit cases, where the sequence points don't come into play because the first expression is able to determine the value of the whole expression on its own.[6]

$$\frac{\tau \text{ is scalar} \quad V_\tau(m_0) = 0 \quad V_{\texttt{signed int}}(m) = 0}{\langle \underline{(m_0, \tau)} \;\texttt{\&\&}\; e, \sigma \rangle \rightarrow_e \langle \underline{(m, \texttt{signed int})}, \sigma \rangle}$$

And for `||`:

$$\frac{\tau \text{ is scalar} \quad V_\tau(m_0) \neq 0 \quad V_{\texttt{signed int}}(m) = 1}{\langle \underline{(m_0, \tau)} \;\texttt{||}\; e, \sigma \rangle \rightarrow_e \langle \underline{(m, \texttt{signed int})}, \sigma \rangle}$$

When the logical operators do not short-circuit, a sequence point is reached, and the evaluation must proceed with the second argument. At this point we need to introduce a new expression form, the constructor $\widehat{\texttt{\&|}}$. We can not simply leave the existing syntax alone and switch to evaluating the second argument if we see that LHS is fully evaluated, because we won't be able to tell whether or not the sequence point has been reached. The rule for `&&` in this situation is

$$\frac{\tau \text{ is scalar} \quad \Pi_\sigma = \emptyset \quad V_\tau(m) \neq 0}{\langle \underline{(m, \tau)} \;\texttt{\&\&}\; e, \sigma_0 \rangle \rightarrow_e \langle \widehat{\texttt{\&|}}(e), \sigma_0[R := \emptyset, \Upsilon := \emptyset] \rangle}$$

The rule for `||` is similar:

$$\frac{\tau \text{ is scalar} \quad \Pi_\sigma = \emptyset \quad V_\tau(m) = 0}{\langle \underline{(m, \tau)} \;\texttt{||}\; e, \sigma_0 \rangle \rightarrow_e \langle \widehat{\texttt{\&|}}(e), \sigma_0[R := \emptyset, \Upsilon := \emptyset] \rangle}$$

The $\mathcal{E}$ and $\mathcal{L}$ contexts are both extended to allow computations to proceed in the argument to $\widehat{\texttt{\&|}}$. We need only specify the rules for $\widehat{\texttt{\&|}}$ when its

[6] The typing conditions are again repeated from the static semantics.

argument is fully evaluated. They are:[7]

$$\frac{\tau \text{ is scalar} \quad V_\tau(m_0) = 0 \quad V_{\texttt{signed int}}(m) = 0}{\langle \widehat{\texttt{\&|}}\,\underline{(m_0, \tau)}, \sigma\rangle \to_e \langle \underline{(m, \texttt{signed int})}, \sigma\rangle}$$

$$\frac{\tau \text{ is scalar} \quad V_\tau(m_0) \neq 0 \quad V_{\texttt{signed int}}(m) = 1}{\langle \widehat{\texttt{\&|}}\,\underline{(m_0, \tau)}, \sigma\rangle \to_e \langle \underline{(m, \texttt{signed int})}, \sigma\rangle}$$

The conditional operator The conditional operator (–?–:–) is ternary. Evaluation begins with the evaluation of its first argument, and then depending on the resulting value, the value of the whole expression is either the second or third argument. This value can not be an lvalue. The type of the result is the same regardless of which argument is chosen, and is calculated according to some relatively complicated rules, which we omit here. We assume that the type τ_c of the result has been calculated already, given the types of the two sub-expressions. The result of the sub-expression's evaluation is cast to this type using the type-cast operator.[8]

$$\frac{\tau \text{ is scalar} \quad \Pi_\sigma = \emptyset \quad V_\tau(m) \neq 0}{\langle \underline{(m, \tau)} \mathbin{?} e_1 : e_2, \sigma\rangle \to_e \langle (\tau_c) e_1, \sigma[R := \emptyset, \Upsilon := \emptyset]\rangle}$$

$$\frac{\tau \text{ is scalar} \quad \Pi_\sigma = \emptyset \quad V_\tau(m) = 0}{\langle \underline{(m, \tau)} \mathbin{?} e_1 : e_2, \sigma\rangle \to_e \langle (\tau_c) e_2, \sigma[R := \emptyset, \Upsilon := \emptyset]\rangle}$$

3.3.4 Side effect operators

An expression that causes side effects will do so through the action of one of a limited set of operators. Before describing these operators, we first describe the way in which side effects alter the contents of memory. Side effects are denoted $\clubsuit(a, m)$, meaning that the value m is to be written to the address a.[9] The variables η, η_1 etc. are also used to denote side effects.

Recall that side effects are not applied immediately upon generation. Instead, they are "queued" in a bag of pending side effects. At any time,

[7] Note that $\widehat{\texttt{\&|}}(e)$ is equivalent to `!!`e, where `!` is the logical negation operator.

[8] In fact, the situation is slightly more complicated than presented here: the arms of conditionals may yield `struct` values, and in this situation one can't cast, but one has to rely on fact that the static semantics will require both arms to have the same `struct` type.

[9] The use of the club suit is meant to suggest that this is something that is about to clobber memory.

the abstract machine can pull a side effect from this bag, and apply it, updating memory with the new value. The rule for this is

$$\frac{\eta \in \Pi_\sigma \quad s \subseteq \Lambda_\sigma \quad R_\sigma \cap s = \emptyset \quad \Upsilon_\sigma \cap s = \emptyset}{\langle e, \sigma \rangle \rightarrow_e \langle e, \sigma[I := I \cup s, M := M[r := m], \Pi := \Pi - \{\eta\}, \Upsilon := \Upsilon \cup s] \rangle}$$

$$\text{where} \quad \begin{array}{rcl} \eta & = & \clubsuit(a, m) \\ r & = & \langle a \ldots a + |m| - 1 \rangle \\ s & = & \{a \ldots a + |m| - 1\} \end{array}$$

This is the only place where memory changes in the entire semantics. Our remaining "side effect operators" will add to the bag of pending side effects Π_σ, and thereby affect memory indirectly. Ultimately those changes must "pass" this rule to have their effect. Before examining those operators, we first consider the possibility that the preconditions in the above rule are not met. Undefinedness is the result of course:

$$\frac{\eta \in \Pi_\sigma \quad (s \not\subseteq \Lambda_\sigma \vee \Upsilon_\sigma \cap s \neq \emptyset \vee R_\sigma \cap s \neq \emptyset) \quad e \neq \mathcal{U}}{\langle e, \sigma \rangle \rightarrow_e \langle \mathcal{U}, \sigma \rangle}$$

$$\text{where} \quad \begin{array}{rcl} \eta & = & \clubsuit(n, m) \\ s & = & \{n \ldots n + |m| - 1\}. \end{array}$$

The final requirement, that $e \neq \mathcal{U}$, is added to stop the model allowing this rule to repeatedly fire once undefinedness is reached.

The assignment operators C defines both a traditional assignment operator, written =, and a number of compound assignments, where the action of the assignment is combined with that of a normal binary operator. The meaning of $e_1 \odot\!= e_2$ is defined to be the same as $e_1 = e_1 \odot e_2$ ($\odot$ a binary operator), except that the expression e_1 is only evaluated once (clearly important in the presence of side effects).

The abstract syntax for assignment expressions of both types needs to include an extra component: a bag containing a record of all the references made to memory in the course of the evaluation of an assignment expression's right hand side. Assignment expressions will thus be written as $e_1 \overset{\beta}{\odot\!=} e_2$, where β is the bag of references to memory, and $\odot$ is the binary operator compounded with the assignment (if any). Furthermore, it is assumed that the bag in an assignment expression is always empty when an evaluation begins.[10]

[10] This sort of constraint is easy to enforce as part of the implementation of the parser which takes the C program's raw syntax, and returns its abstract syntax tree.

Finally, it is assumed that the right-hand side of the expression will be wrapped in an `RVR` constructor. This ensures not only that the expression there will decay to a value, but also that it will do so in a way that the rules for assignment will be able to "see". Recall that the transformation of lvalues into values (the $\leadsto$ relation) is where the R component of a state may increase. This increase corresponds to referring to an object in memory, and is exactly what must be recorded in the assignment syntax's bag.

A naïve version of the semantics might do away with the R and extend the $\mathcal{L}$ to include the right-hand sides of assignment expressions. Unfortunately, this would allow the following transition, which leaves the bag β unchanged, to take place:

$$\frac{\overline{\langle \mathtt{LV}(a,t), \sigma_0\rangle \leadsto \langle m, \sigma\rangle}\ \leadsto}{\langle e_1 \overset{\beta}{\odot=} \mathtt{LV}(a,t), \sigma_0\rangle \to_e \langle e_1 \overset{\beta}{\odot=} m, \sigma\rangle}\ \mathcal{L}\text{ context}$$

The correct rule controls the evaluation of the right hand side and monitors the way in which references to memory are made. Allowing the naïve version of the rules would give two rules scope over the situation, and the monitoring of references could be bypassed, as in the above example.

The first rule we present allows for the expansion of the abbreviation implicit in the compound assignment. This expansion takes $e_1 \overset{\beta}{\odot=} e_2$ to $e_1 \overset{\beta}{=} e_1 \odot e_2$ whenever e_1 is an lvalue, meaning it will not admit any further reductions:

$$\frac{}{\langle \mathtt{LV}(a,\tau) \overset{\beta}{\odot=} e_2, \sigma_0\rangle \to_e \langle \mathtt{LV}(a,\tau) \overset{\beta}{=} \mathtt{LV}(a,\tau) \odot e_2, \sigma_0\rangle}$$

In this situation, it doesn't matter whether or not the RHS of the assignment has lost its `RVR` wrapper, as the evaluation of a binary operator sub-tree must always result in a value.

The next rule for assignment controls the way in which evaluation of the assignment's right hand side proceeds (the rule for simple assignment is identical):

$$\frac{\langle e_2, \sigma_0\rangle \to_e \langle e_2', \sigma\rangle}{\langle e_1 \overset{\beta}{\odot=} e_2, \sigma_0\rangle \to_e \langle e_1 \overset{\beta'}{\odot=} e_2', \sigma\rangle}$$

where $\beta' = \beta + (R_\sigma - R_{\sigma_0}) - (R_{\sigma_0} - R_\sigma)$.

The expression for determining β' mimics the changes to the R component of the state. Thus, if the evaluation of the right hand side e_2 makes additional references β_1 such that $R_\sigma = R_{\sigma_0} + \beta_1$, then β' will equal $\beta + \beta_1$.

However, it is also possible that the reference maps of program states will decrease. This will happen every time a sequence point is encountered, for example. The β component of the assignment syntax is therefore a bag of references to memory made in the course of the evaluation of the right hand side, and which are still current.

The motivation for keeping track of these references is revealed in the next rule. Previously (in section 3.1.2) it was noted that the abstract machine is allowed to make references to memory that may seem to clash with an update, if the references were made in the service of calculating the updated object's new value. This is precisely what happens in an expression such as `i = i + 1`. This "permission to refer" is implemented by removing the appropriate references when the assignment takes place. The rule is

$$\frac{\langle (\tau)\,\underline{(m_0,\tau_0)}, \sigma_0\rangle \to_e \langle v, \sigma_0\rangle \quad \tau \text{ not an array type}}{\langle \mathtt{LV}(a,\tau) \stackrel{\beta}{=} \underline{(m_0,\tau_0)}, \sigma_0\rangle \to_e \langle v, \sigma\rangle}$$

where $\sigma = \sigma_0[R := R - \beta\langle a \ldots a + |\tau_0| - 1\rangle, \Pi := \Pi + \{\clubsuit(a, m)\}]$
The value $\underline{(m,\tau)}$ is found by using the cast operator to convert the value on the RHS of the assignment to the type of the object on the LHS. The state σ_0 is modified by removing references in β to memory corresponding to the object assigned from the reference map, and by adding the updating side effect to those pending in σ_0.

Now, the question that naturally arises when confronted with these last two, ugly assignment rules is whether or not they conform to the natural language of "references to the object assigned are allowed on the right hand side". There are two criteria to assess: the rules should not make things undefined that are defined in the standard, and *vice versa*, they should not give meaning to things that are undefined.

Consider the first case. The only way in which these rules might make something incorrectly undefined would be if an allowable reference in the bag β were to clash with an attempt to update the same object elsewhere, thereby running afoul of the rule forbidding reference and update of the same object. However, if such an update takes place, then the expression is undefined anyway, because the allowable reference is only a prelude to another update of the same object which will occur whenever the side effect generated by the assignment is applied. This has to cause undefined behaviour because two updates of the same object are also illegal. So, the rules will not make a difference in this way.

We must also consider the possibility that the two rules might give meaning to something which is actually undefined. This can not be, as our method of emulating what is required actually records references that shouldn't be recorded. This can only make things worse as far as defined-

ness is concerned.

Post increment and decrement The rules for the post-increment (++) and post-decrement (--) operators differ only slightly. Although the rule does look at the value of the object, it does not record this as a reference to the object in R_σ. This is because, as with the assignment, this is clearly a reference made in order to calculate the fresh value. After pulling the fresh value out of memory, all that is required is to make the new value a pending side effect. So, the rule for ++ is

$$\frac{s \subseteq I_\sigma \quad v \neq \mathcal{U}}{\langle \texttt{LV}(a,\tau)\texttt{++}, \sigma\rangle \rightarrow_e \langle \underline{(m,\tau)}, \sigma[\Pi := \Pi + \{\clubsuit(a,v)\}]\rangle}$$

$$\begin{array}{llcl} \text{where} & V_{\texttt{signed int}}(m') & = & 1 \\ & v & = & \texttt{OpSem}(\sigma, \texttt{+}, \underline{(m,\tau)}, \underline{(m', \texttt{signed int})}) \\ & m & = & M_\sigma\langle a \ldots a + |\tau| - 1\rangle \\ & s & = & \{a \ldots a + |\tau| - 1\} \end{array}$$

The rule for `--` is just the same except that `++` is replaced by `--`, and the operator used in the call to `OpSem` is `-`. Undefined behaviour is possible in two ways that need to be given their own rule: the object to be incremented might not be initialised, or the increment operation may fail.

$$\frac{s \not\subseteq I_\sigma \vee v = \mathcal{U}}{\langle \texttt{LV}(a,\tau)\texttt{++}, \sigma\rangle \rightarrow_e \langle \mathcal{U}, \sigma\rangle}$$

$$\begin{array}{llcl} \text{where} & V_{\texttt{signed int}}(m') & = & 1 \\ & v & = & \texttt{OpSem}(\sigma, \texttt{+}, \underline{(m,\tau)}, \underline{(m', \texttt{signed int})}) \\ & m & = & M_\sigma\langle a \ldots a + |\tau| - 1\rangle \\ & s & = & \{a \ldots a + |\tau| - 1\} \end{array}$$

Finally, recall that pre-increment (`++x`) and pre-decrement (`--x`) are both handled by assuming translation to the equivalent `x += 1` and `x -= 1`.

3.3.5 Function calls—interfacing with statements

The final form of expression is the function call. There are two rules for this as there is a sequence point after the arguments and the function designator are evaluated. The $\mathcal{E}$ context ensures that these "inner" evaluations take place, so the following rule merely records the fact that the sequence point has been reached. Like the logical boolean operations `&&` and `||`, this requires the addition of a new intermediate syntactic form, which we write

$\hat{f}$, for function reference values f.

$$\frac{\Pi_\sigma = \emptyset \quad \text{all of the } e_i\text{s are values} \quad V_\tau(m) \neq 0}{\langle \underline{(m,\tau)}(e_1,\ldots,e_n),\sigma\rangle \to_e \langle \widehat{\underline{(m,\tau)}}(e_1,\ldots,e_n),\sigma[R := \emptyset, \Upsilon := \emptyset]\rangle}$$

where $\tau = \tau_1 \times \ldots \tau_n \to \tau_r$

If the function expression evaluated to the null function reference, then the program is undefined:

$$\frac{V_\tau(m) = 0}{\langle \underline{(m,\tau)}(e_1,\ldots,e_n),\sigma\rangle \to_e \langle \mathcal{U},\sigma\rangle}$$

The call to the function is evaluated in this final expression rule:

$$\frac{\langle s, \texttt{inst_parms}(\sigma_0, \mathit{args}, [e_1 \ldots e_n])\rangle \to_s \langle v_s, \sigma\rangle}{\langle \widehat{\underline{(m,\tau)}}(e_1,\ldots,e_n),\sigma_0\rangle \to_e \langle v, \sigma_0[M := M_\sigma, I := I_\sigma \cap \Lambda_{\sigma_0}\rangle}$$

where
$$\begin{array}{rcl} v & = & \begin{cases} \underline{(m,\tau_r)} & \text{if } v_s = \texttt{RetVal}\underline{(m,\tau_r)} \\ \underline{\mathcal{U}} & \text{if } v_s = \mathcal{U} \end{cases} \\ V_\tau(m) & = & F \\ \Phi_{\sigma_0}(F) & = & (\mathit{args}, s, \tau) \end{array}$$

Note that the final state is the same as the initial state except for the contents of memory and the initialisation map.[11] The $\to_s$ relation is the statement evaluation relation, and the `RetVal` wraps a statement evaluation's return value. Both of these constructions are described in the next section, which explains the semantics of statements.

The Φ component of the state returns information (names and types) about a function's logical arguments as well as the function's body and overall type. This information is then used by the `inst_parms` function to install the argument values (e_1 to e_n) in memory, and to update the environment information stored in the state. We define it recursively over

[11] If we were modelling heap allocation of memory, we would need to allow for changes to the allocation map (Λ).

parameter lists thus:

$$\begin{array}{l}
\texttt{inst_parms}(\sigma, (\mathsf{id}, \tau) :: \textit{tail}, (v_0, \tau_0) :: vs) = \\
\quad \texttt{inst_parms}\left(\sigma\left[\begin{array}{l} A := A^g[\mathsf{id} := a], \\ \Gamma := \Gamma^g[\mathsf{id} := \tau], \\ I := I \cup r, \\ \Lambda := \Lambda \cup r, \\ M := M[\langle a \ldots \rangle := C(v_0)], \\ \Sigma := \Sigma^g \end{array}\right], \textit{tail}, vs\right) \\
\quad \text{where } \texttt{alloc}(\sigma, \tau, a) \\
\qquad\qquad r = \{a \ldots a + |\tau| - 1\} \\
\qquad\qquad C : \tau_0 \to \tau \text{ is the conversion function from } \tau_0 \text{ to } \tau
\end{array}$$

The base case is

$$\texttt{inst_parms}(\sigma, [], []) = \sigma$$

The `alloc` relation is true of its three arguments if the third, a, is the address of a piece of unallocated memory suitable for use by the given type. If there is no such address available, then undefinedness results. The actual definition is thus a slightly more complicated version of that presented above, as this possibility must be allowed for.

The called function's new environment is constructed by building upon the basis of the global environment (denoted by the ^{g} superscript). Building on the existing environment's maps would give our language dynamic scoping.

This definition of function calls does not admit the possibility that function bodies might execute in parallel. This might be supposed to happen in an expression such as `f(x) + g(y)`. This is again a matter of some controversy among those concerned with interpreting the standard. Ultimately the decision was made to disallow call interleaving because it allows us to use a big-step semantics for statement execution (though this has other problems). In fact, given the lack of language one way or the other on this question in the standard, it would be reasonable to suppose that the model should not be constrained so as to forbid interleaving.

3.4 Statement execution

C's statements are rather fewer in number than its expressions, and the rules are also rather simpler. In particular, it is possible to write the rules using a big-step style. The statement relation $\to_s$ maps statement-state

pairs to *statement value*-state pairs. A statement value is one of the following limited set of possibilities, each describing how statement execution has come to finish at this point.

`BreakVal`	a break statement was encountered
`ContVal`	a continue statement was encountered
`RetVal`$(\underline{m, \tau})$	a return statement (with value) was encountered
`StmtVal`	an ordinary evaluation termination

All but `StmtVal` interfere with the statement sequencing rule. There is also an undefined value, represented as before by $\mathcal{U}$. We will use v to vary over statement values. We also use $\rightarrow_s$ to denote the evaluation relation over lists of statements. This extended relation has the same range of possible result types.

The choice of a big-step semantics for the definition of statement semantics has its problems. In a simpler language than C, a typical big-step definition for loops might be (using $\top$ and $\bot$ for the boolean values):

$$\frac{G, \sigma_0 \Downarrow_e \top \quad \langle S\,;\, \textbf{while}\ G\ \textbf{do}\ S,\ \sigma_0\rangle \Downarrow_s \sigma}{\langle \textbf{while}\ G\ \textbf{do}\ S,\ \sigma_0\rangle \Downarrow_s \sigma}$$

$$\frac{G, \sigma \Downarrow_e \bot}{\langle \textbf{while}\ G\ \textbf{do}\ S,\ \sigma\rangle \Downarrow_s \sigma}$$

If this language is deterministic, then we can express the question of whether or not a program S terminates starting in state σ_0, as

$$\exists \sigma.\ \langle S,\ \sigma_0\rangle \Downarrow_s \sigma$$

If the language is non-deterministic then we have two different notions of termination: whether or not the program is guaranteed to terminate, and whether or not it *might* terminate. (Alternatively, we have a trichotomy: either a program must terminate, or it must diverge, or it may be able to both loop and terminate.) In this case, the expression above only puts the question of whether or not the program *might* terminate (alternatively, whether or not it must diverge).

A typical way of defining the more useful “must terminate” notion is to define another relation over the syntax to embody this idea. We dislike this approach because it is impossible to relate the new relation to the old; one has to decide by inspection that the relation is correct. Our intertwined evaluation relations are complicated enough on their own, without adding another relation to the brew. We rather add a rule which sends a loop to the undefined return value if there is an infinite evaluation sequence starting from the starting state of the loop. (Bofinger’s semantics [Bof98], though

deterministic, also has such a rule.) Our rule, in section 3.4.5, ensures that only very few statement evaluations ever block,[12] and we thus capture the concept of statement s being sure to terminate starting in state σ_0 by using the following expression:

$$\forall v, \sigma.\ \langle s, \sigma_0 \rangle \rightarrow_s \langle v, \sigma \rangle \Rightarrow (v = \texttt{StmtVal})$$

This formulation of the semantics for statements does have the disadvantage of precluding a proper examination of programs that exhibit infinite behaviours. If we used a small-step system for statements as well as for expressions, we could do this. However, the advantage of the big-step approach is that one has an easy definition of what happens when new blocks and environments are entered and exited. Arranging for just the right variables to fall out of scope as a block exits is much more complicated in a small-step presentation.

3.4.1 Simple statements

Empty statements The first statement evaluation rule is that for the empty statement, written here just using the `;`.

$$\frac{}{\langle \texttt{;}, \sigma \rangle \rightarrow_s \langle \texttt{StmtVal}, \sigma \rangle}$$

Expression statements An expression statement consists only of an expression, which is evaluated for the side effects caused. Note the use of RVR, which gives the expression a valid $\mathcal{L}$ context in which to evaluate values which may be lvalues, and the resetting of the state's "side effect components", Π, R, and Υ.

$$\frac{\langle \texttt{RVR}(e), \sigma_0[\Pi := \emptyset, R := \emptyset, \Upsilon := \emptyset] \rangle \rightarrow_e^* \langle \underline{(m, \tau)}, \sigma \rangle \quad \Pi_\sigma = \emptyset}{\langle e\texttt{;}, \sigma_0 \rangle \rightarrow_s \langle \texttt{StmtVal}, \sigma \rangle}$$

If the expression evaluation goes astray we promote the expression undefinedness to the level of statements:

$$\frac{\langle \texttt{RVR}(e), \sigma_0[\Pi := \emptyset, R := \emptyset, \Upsilon := \emptyset] \rangle \rightarrow_e^* \langle \mathcal{U}, \sigma \rangle}{\langle e\texttt{;}, \sigma_0 \rangle \rightarrow_s \langle \mathcal{U}, \sigma \rangle}$$

[12] The problematic cases are functions that recurse indefinitely without requiring the allocation of any space in memory for parameters or local variables. Such a function is `int f(void) { return f(); }`. With local variables or parameters present, recursing functions will eventually cause undefinedness in our model because they will run out of memory. This failing in the model could be addressed by arbitrarily requiring that each function invocation allocated a little space in memory, but this is not a pleasant prospect.

3.4.2 Interruptions

The `break`, `continue` and `return` statements all interrupt the normal flow of control. Collectively we shall refer to them as *interruption statements*.

$$\frac{}{\langle \texttt{break;}, \sigma\rangle \rightarrow_s \langle \texttt{BreakVal}, \sigma\rangle}$$

$$\frac{}{\langle \texttt{continue;}, \sigma\rangle \rightarrow_s \langle \texttt{ContVal}, \sigma\rangle}$$

$$\frac{\langle \texttt{RVR}(e), \sigma_0[\Pi := \emptyset, R := \emptyset, \Upsilon := \emptyset]\rangle \rightarrow_e^* \langle m, \sigma\rangle \quad \Pi_\sigma = \emptyset}{\langle \texttt{return}\ e\texttt{;}, \sigma_0\rangle \rightarrow_s \langle \texttt{RetVal}(m), \sigma\rangle}$$

$$\frac{}{\langle \texttt{return;}, \sigma\rangle \rightarrow_s \langle \texttt{RetVal}\underline{(\emptyset, \texttt{void})}, \sigma\rangle}$$

There are two rules for `return` because it exists in two forms, both with a value to be returned and without. In the latter case, we use the $\emptyset$ symbol to stand for a null (empty) value. No well-behaved program will attempt to make use of such a value, as it will only be returned by functions returning `void`.[13]

3.4.3 Compound statements

Blocks Statements can be grouped together in a block. A block consists of a list of variable declarations followed by a list of statements. When the statements of a block finish evaluating, they will do so in an environment different from the external one. Just as happened with the rule for function calls, this is rectified as the block exits by updating the original state with the memory information from the new state.

$$\frac{\langle \textit{decls}, \sigma_0\rangle \rightarrow_v \langle \texttt{VarDeclVal}, \sigma_1\rangle \quad \langle \textit{stmts}, \sigma_1\rangle \rightarrow_s \langle v, \sigma_2\rangle}{\langle \{\textit{decls stmts}\}, \sigma_0\rangle \rightarrow_s \langle v, \sigma_0[M := M_{\sigma_2}, I := I_{\sigma_2} \cap \Lambda_{\sigma_0}]\rangle}$$

Even variable declarations are not immune to failure, so that the following rule is also necessary:

$$\frac{\langle \textit{decls}, \sigma_0\rangle \rightarrow_v \langle \mathcal{U}, \sigma\rangle}{\langle \{\textit{decls stmts}\}, \sigma_0\rangle \rightarrow_s \langle \mathcal{U}, \sigma\rangle}$$

The $\rightarrow_v$ relation gives meaning to variable declarations. We again abuse our notation slightly to let it stand for the relation over lists of declarations. These relations and the `VarDeclVal` value are further explained in section 3.5.

[13] This is not strictly true: the standard actually allows `return` statements without expressions to appear anywhere. They will cause the flow of control to return to the calling expression, but the “value” returned can not be used.

Statement sequencing A list of statements, as occurs in a block, is executed in order, with the requirement that for execution to continue, the statement value of the last statement executed must have been StmtVal. The base case is the empty list:

$$\frac{}{\langle [\,], \sigma\rangle \rightarrow_s \langle \texttt{StmtVal}, \sigma\rangle}$$

A normal evaluation proceeds according to the following rule:

$$\frac{\langle s_1, \sigma_0\rangle \rightarrow_s \langle \texttt{StmtVal}, \sigma_1\rangle \quad \langle \mathit{stmts}, \sigma_1\rangle \rightarrow_s \langle v, \sigma\rangle}{\langle s_1 :: \mathit{stmts}, \sigma_0\rangle \rightarrow_s \langle v, \sigma\rangle}$$

On the other hand, an interrupted evaluation (here an undefined result is also an interruption) will look like

$$\frac{\langle s_1, \sigma_0\rangle \rightarrow_s \langle v, \sigma\rangle \quad v \neq \texttt{StmtVal}}{\langle s_1 :: \mathit{stmts}, \sigma_0\rangle \rightarrow_s \langle v, \sigma\rangle}$$

This means that interruption statements will cause all further statements in a sequential composition to be skipped.

3.4.4 Conditional statements

There are three rules for the `if` statement. The first copes with the failure of the guard to evaluate properly.

$$\frac{\langle \texttt{RVR}(e), \sigma_0[\Pi := \emptyset, R := \emptyset, \Upsilon := \emptyset]\rangle \rightarrow_e^* \langle \mathcal{U}, \sigma\rangle}{\langle \texttt{if (}e\texttt{)}\ s_1\ \texttt{else}\ s_2, \sigma_0\rangle \rightarrow_s \langle \mathcal{U}, \sigma\rangle}$$

The two rules where the expression does actually evaluate fully are entirely straightforward. If the guard expression evaluates to a non-zero value then the first branch is evaluated, and its statement value preserved.

$$\frac{\langle \texttt{RVR}(e), \sigma_0[\Pi := \emptyset, R := \emptyset, \Upsilon := \emptyset]\rangle \rightarrow_e^* \langle \underline{(m, \tau)}, \sigma_1\rangle \quad \langle s_1, \sigma_1\rangle \rightarrow_s \langle v, \sigma\rangle}{\langle \texttt{if (}e\texttt{)}\ s_1\ \texttt{else}\ s_2, \sigma_0\rangle \rightarrow_s \langle v, \sigma\rangle}$$

where τ is scalar, $V_\tau(m) \neq 0$, and $\Pi_{\sigma_1} = \emptyset$
Otherwise, the second branch is chosen:

$$\frac{\langle \texttt{RVR}(e), \sigma_0[\Pi := \emptyset, R := \emptyset, \Upsilon := \emptyset]\rangle \rightarrow_e^* \langle \underline{(m, \tau)}, \sigma_1\rangle \quad \langle s_2, \sigma_1\rangle \rightarrow_s \langle v, \sigma\rangle}{\langle \texttt{if (}e\texttt{)}\ s_1\ \texttt{else}\ s_2, \sigma_0\rangle \rightarrow_s \langle v, \sigma\rangle}$$

where τ is scalar, $V_\tau(m) = 0$, and $\Pi_{\sigma_1} = \emptyset$

3.4.5 Iteration

There are three looping constructions in C, the `while` loop, the `for` loop, and the `do-while` loop. We model them all with two special intermediate syntactic forms, the O (loop) and T (trap) constructors. The first is the basis for a generic looping mechanism, and the second is a mechanism which allows interrupt values to be intercepted or "trapped". The O constructor takes the loop guard and the loop body as arguments, while the T constructor takes the interrupt value to be intercepted and the statement which will be executed. The translations for the three loop forms in C are

$$\begin{array}{rcl} \texttt{while (}g\texttt{)}\ s & \hat{=} & T(\texttt{BreakVal}, O(g, T(\texttt{ContVal}, s))) \\ \texttt{for (}e_1\texttt{; }e_2\texttt{; }e_3\texttt{)}\ s & \hat{=} & \{e_1\texttt{;}\ T(\texttt{BreakVal}, O(e_2, \{T(\texttt{ContVal}, s)\ \ e_3\texttt{;}\}))\} \\ \texttt{do}\ s\ \texttt{while (}g\texttt{);} & \hat{=} & T(\texttt{BreakVal}, \{T(\texttt{ContVal}, s)\ \ O(g, T(\texttt{ContVal}, s))\}) \end{array}$$

Note that the juxtapositioning of elements inside a pair of braces above indicates sequencing; the semi-colon is used as a statement terminator, not as a statement separator. The rule for the `for` means that while a continue statement in the loop body may interfere with the rest of the loop body, it will not prevent the third expression from being evaluated. The syntax for the `for` loop also allows expressions to be omitted. We model this case by inserting the expression 1 in the place of the omitted expression. This has no side effects, but the fact that it is not zero will cause the loop to keep looping. (The statement `for (;;) s` is an idiomatic way of writing either an infinite loop, or one that will exit due to a `break` or `return` statement.)

There are two rules for the T construction. If the value returned by the statement wrapped up is the one trapped, then a `StmtVal` is returned instead of what was going to be returned.[14]

$$\frac{\langle s, \sigma_0 \rangle \rightarrow_s \langle v, \sigma \rangle}{\langle T(v, s), \sigma_0 \rangle \rightarrow_s \langle \texttt{StmtVal}, \sigma \rangle}$$

Otherwise, the return value is passed through unchanged.

$$\frac{\langle s, \sigma_0 \rangle \rightarrow_s \langle v, \sigma \rangle \quad v' \neq v}{\langle T(v', s), \sigma_0 \rangle \rightarrow_s \langle v, \sigma \rangle}$$

There are five rules for O. The first two specify the behaviour when the guard expression doesn't evaluate to true, which can happen in two different ways. The guard might result in undefined behaviour, in which case

[14] There are very clear parallels here with the way in which exceptions in a language such as SML are caught; here the exceptions are very simple, atomic values.

the loop's behaviour is also undefined.

$$\frac{\langle \texttt{RVR}(g), \sigma_0[\Pi := \emptyset, R := \emptyset, \Upsilon := \emptyset]\rangle \rightarrow_e^* \langle \mathcal{U}, \sigma\rangle}{\langle O(g,s), \sigma_0\rangle \rightarrow_s \langle \mathcal{U}, \sigma\rangle}$$

Alternatively, the loop guard may evaluate to a zero value:

$$\frac{\langle \texttt{RVR}(g), \sigma_0[\Pi := \emptyset, R := \emptyset, \Upsilon := \emptyset]\rangle \rightarrow_e^* \langle \underline{(m,\tau)}, \sigma\rangle}{\langle O(g,s), \sigma_0\rangle \rightarrow_s \langle \texttt{StmtVal}, \sigma\rangle}$$

where τ is a scalar type, $V_\tau(m) = 0$, and $\Pi_\sigma = \emptyset$

If, however, the guard does evaluate to a non-zero value, then the loop is entered. The first rule below covers those cases where the body doesn't evaluate to a `StmtVal`; this causes the loop to exit. Consider then how this rule interacts with the translations of the standard C forms; if a `continue` statement is encountered while evaluating a loop body, then this will have been trapped by the T that is always wrapped around the occurrences of the body in the translation. Thus, this rule will not apply. In the case of the `break` statement, this rule *will* apply, but the `BreakVal` will be trapped by the trap around the entirety of the loop.

$$\frac{\langle \texttt{RVR}(g), \sigma_0[\Pi := \emptyset, R := \emptyset, \Upsilon := \emptyset]\rangle \rightarrow_e^* \langle \underline{(m,\tau)}, \sigma_1\rangle \quad \langle s, \sigma_1\rangle \rightarrow_s \langle v, \sigma\rangle}{\langle O(g,s), \sigma_0\rangle \rightarrow_s \langle v, \sigma\rangle}$$

where $v \neq \texttt{StmtVal}$, τ is a scalar type, $V_\tau(m) \neq 0$, and $\Pi_{\sigma_1} = \emptyset$.

If the loop body's execution does terminate normally, the loop is entered once more:

$$\frac{\begin{array}{c}\langle \texttt{RVR}(g), \sigma_0[\Pi := \emptyset, R := \emptyset, \Upsilon := \emptyset]\rangle \rightarrow_e^* \langle \underline{(m,\tau)}, \sigma_1\rangle \\ \langle s, \sigma_1\rangle \rightarrow_s \langle \texttt{StmtVal}, \sigma_2\rangle \qquad \langle O(g,s), \sigma_2\rangle \rightarrow_s \langle v, \sigma\rangle\end{array}}{\langle O(g,s), \sigma_0\rangle \rightarrow_s \langle v, \sigma\rangle}$$

where τ is a scalar type, $V_\tau(m) \neq 0$, and $\Pi_{\sigma_1} = \emptyset$.

The rule that captures the behaviour of possible infinite loops is as follows:

$$\frac{\begin{array}{l}\forall n. \exists m, \sigma'. \\ \quad \langle \texttt{RVR}(g), f(n)[\Pi := \emptyset, R := \emptyset, \Upsilon := \emptyset]\rangle \rightarrow_e^* \langle \underline{(m,\tau)}, \sigma'\rangle \wedge \\ \quad (V_\tau(m) \neq 0) \wedge (\Pi_{\sigma'} = \emptyset) \wedge \langle s, \sigma'\rangle \rightarrow_s \langle \texttt{StmtVal}, f(n+1)\rangle \\ f(0) = \sigma_0\end{array}}{\langle O(g,s), \sigma_0\rangle \rightarrow_s \langle \mathcal{U}, \sigma_0\rangle}$$

where $f : \mathbb{N} \rightarrow \texttt{CState}$

3.5 Variable declarations

Variables can be declared in two possible contexts, either at the start of a block, in which case they are by default *automatic* variables (which will cease to exist in any meaningful sense after the block exits), or they can be declared at the top level, in which case they are *static*, and have lifetimes equal to the duration of the program. Recall that it is also possible to declare static variables inside a block, but this possibility is one that we side-step. Instead we model static local variables as global variables with distinct names, and that only occur in the block where the variable is declared.

Upon declaration, variables can also be initialised. This naturally involves use of the $\rightarrow_e$ relation. Finally, though not strictly a variable declaration, structure declarations can also occur wherever a variable declaration is permitted. The $\rightarrow_v$ relation takes declaration-state pairs and returns value-state pairs. There are only two possible return values for variable declarations, `VarDeclVal`, representing a successful execution, and $\mathcal{U}$ for undefinedness. The rule for a successful variable declaration is:

$$\frac{}{\langle \tau\ \mathsf{id}\,\texttt{;}, \sigma\rangle \rightarrow_v \langle \texttt{VarDeclVal}, \texttt{decl_var}(\sigma, \mathsf{id}, \tau)\rangle}$$

The `decl_var` function updates the state argument's environment, allocating the variable in question some space in memory (space appropriate for its type), and changing the address, allocated and type maps in the state (A, Λ and Γ respectively).

$$\texttt{decl_var}(\sigma, \mathsf{id}, \tau) = \sigma \begin{bmatrix} A := A[\mathsf{id} := a], \\ \Gamma := \Gamma[\mathsf{id} := \tau], \\ \Lambda := \Lambda \cup r \end{bmatrix}$$
$$\text{where } \texttt{alloc}(\sigma, \tau, a) \text{ and } r = \{a \ldots a + |\tau| - 1\}$$

Though we have omitted the rule here, if there is no memory available for `alloc` then the declaration will fail with undefinedness the result.

When a variable is initialised with an expression it is equivalent to first declaring the variable, and then assigning the initialising expression to it.

$$\frac{\begin{array}{c}\langle \tau\ \mathsf{id}\,\texttt{;}, \sigma_0\rangle \rightarrow_v \langle \texttt{VarDeclVal}, \sigma'\rangle \\ \langle \mathsf{id} = \texttt{RVR}(e), \sigma'[R := \emptyset, \Upsilon := \emptyset]\rangle \rightarrow_e^* \langle \underline{(m, \tau)}, \sigma\rangle \quad \Pi_\sigma = \emptyset\end{array}}{\langle \tau\ \mathsf{id}\ \texttt{=}\ e\,\texttt{;}, \sigma_0\rangle \rightarrow_v \langle \texttt{VarDeclVal}, \sigma\rangle}$$

Expression evaluation can go astray, so undefined behaviour can result:

$$\frac{\langle \tau\ \mathsf{id}\,\texttt{;}, \sigma_0\rangle \rightarrow_v \langle \texttt{VarDeclVal}, \sigma'\rangle \quad \langle \mathsf{id} = \texttt{RVR}(e), \sigma'\rangle \rightarrow_e^* \langle \mathcal{U}, \sigma\rangle}{\langle \tau\ \mathsf{id}\ \texttt{=}\ e\,\texttt{;}, \sigma_0\rangle \rightarrow_v \langle \mathcal{U}, \sigma\rangle}$$

Finally, the third type of declaration is for declaring new `struct` types. This requires both a name for the type, and the names and types for all of the constituent fields. This information is stored in the Σ component of the state record.

$$\frac{}{\langle \texttt{struct}\ \mathsf{id}\ \textit{fields};, \sigma\rangle \rightarrow_v \langle \texttt{VarDeclVal}, \sigma[\Sigma := \Sigma[\mathsf{id} := \textit{fields}]]\rangle}$$

Finally, there are three obvious rules for dealing with lists of variable declarations, as occur at the head of blocks.

$$\frac{}{\langle [], \sigma\rangle \rightarrow_v \langle \texttt{VarDeclVal}, \sigma\rangle}$$

$$\frac{\langle vd, \sigma_0\rangle \rightarrow_v \langle \texttt{VarDeclVal}, \sigma'\rangle \quad \langle vs, \sigma'\rangle \rightarrow_v \langle v, \sigma\rangle}{\langle vd :: vs, \sigma_0\rangle \rightarrow_v \langle v, \sigma\rangle}$$

$$\frac{\langle vd, \sigma_0\rangle \rightarrow_v \langle \mathcal{U}, \sigma\rangle}{\langle vd :: vs, \sigma_0\rangle \rightarrow_v \langle \mathcal{U}, \sigma\rangle}$$

3.6 Mechanisation

The mechanisation of C's dynamic semantics in HOL was a long and error-prone process. Here we describe how that mechanisation was performed, touching on the technology used to achieve it as well as some of the design decisions that had to be made. We also describe the ways in which the mechanisation falls short of being a perfect realisation of the definition above.

The centre-piece of the Cholera definition of the dynamic semantics is the mutually recursive, inductive definition of `meaning`. This consists of some sixty rules. We model the $\mathcal{E}$ and $\mathcal{L}$ contexts as constrained functions of type `:CExpr->CExpr`. These functions take expressions to larger expressions that include the domain element somewhere in their body in place of the context's "hole". Without the rules for these contexts, the number of rules in the `meaning` relation would have ballooned.

There are six relations embodied by all of these rules. They are $\rightarrow_e$ and $\rightarrow_e^*$ (this has to be given an explicit definition because it is used inside the definition of $\rightarrow_s$); $\rightarrow_s$ and its extension to lists of statements; and $\rightarrow_v$ and its extension to lists of declarations. The links between the various relations occur at the following points: expression evaluation depends on statement execution in the rule for function calls, statement execution depends on variable declarations inside the rule for blocks, variable declarations depend on expression evaluation because of the possibility of declared

variables being initialised, and statement execution depends on expression evaluation in the expression-statement, and the conditional and iteration forms.

The HOL system's standard inductive definition package does not allow one to define mutually recursive relations explicitly. Instead one constructs one large relation that embodies all of the required forms, distinguishing among them with tags attached to disjoint union types constructed for the purpose. This technique is sufficiently well-understood that its use represents no more than a minor inconvenience to the user.

The main source of errors in this part of the mechanisation was the fact that the definition package used encourages one to enter the various components of an inductive rule (hypotheses, side-conditions, and conclusion) as separate terms. This is superficially appealing at a syntactic level. Instead of writing $P \wedge Q \wedge R$, one is encouraged to construct a list of terms at the level of ML: `[``P``, ``Q``, ``R``]`. The semantics of both input styles is the same, but the latter leads to more errors because type-checking happens many times over disjoint terms instead of just once over a combined term. In this way, variables intended to be the same in two terms can be assigned different types because of an error that would have been caught if the terms had been checked as one. These are errors where the user's clear intent has become corrupted in being transferred to the machine. They and a number of similar errors would be easily caught by an analysis typically done by Prolog systems: flagging variables in the antecedent of a rule that are used only once.

Another class of errors is that where the user's intention is mistaken. These errors are typically harder to catch as the user's misunderstanding may only come to light when a theorem fails to "come out". In Cholera, errors of this sort could persist for a surprisingly long time as proofs involving all of `meaning`'s 60 rules were rare. Rules in error might still satisfy expected properties, so the process of iterative improvement to the definition could and did take a large number of iterations.

Before defining the `meaning` relation, it was necessary to define the type of program states. The various "map" components of the state (e.g., Γ, A) are modelled as functions. Strictly, one might expect these components to be modelled as partial, finite functions, but HOL provides no built-in support for such types and we soon saw that overriding arbitrary total functions with new information would be enough, as well-typed programs would never look beyond the finite amount of new information that had been overlaid onto the functions.

The mechanisation of `meaning` is also dependent on the definitions of the `OpSem` and `UnOpSem`, which give meaning to the various operators which

are part of the language, such as those for performing arithmetic. These definitions are not given above, but *are* provided in the mechanisation.

Our definition of the `meaning` relation differs from that given above in omitting the rule with the infinitary hypothesis that allows for infinite loop detection. This omission was initially caused by the fact that Melham's inductive definition package [Mel91] can not handle such definitions. We later became aware of Harrison's alternative package [Har95b], which does allow this form of rule, and Bofinger's example [Bof98] makes it clear that this is indeed a plausible addition. However, as pointed out earlier, even this form of definition has its flaws, and we should ultimately prefer to recast the definition of statement evaluation to use a small-step relation.

Chapter 4

Expressions

We begin by describing a suite of results which are expected consequences of our definition, and which thereby serve to validate it. The most important of these is the property of type preservation or subject reduction. *We then discuss* expression purity*, a syntactic characterisation of what it is for an expression to be side effect free. We demonstrate that the syntactic property has the desired semantic property and then show that syntactically pure expressions are deterministic. Finally we conclude with the result that sequence point free expressions are also deterministic.*

4.1 Preliminaries

By virtue of having been accepted by the theorem prover (specifically Melham's inductive definition package [Mel91]), our definition of C's dynamic semantics is automatically a valid inductive definition. In other words, the implicit function over sets whose least fixed point we have taken is necessarily monotonic with respect to $\subseteq$ (the Knaster-Tarski result then justifies the fix-point itself). This is the most minimal of all possible "sanity checks", and tells us nothing about the quality of our semantics with respect to the original specification, the ISO standard [ISO90].

A first attempt at better validation for a semantics such as ours consists of simple eye-balling by the author. This is an error-prone process in itself, but it can catch simple mistakes. Better would be to have the specification of the semantics inspected by another individual who was both familiar with the fine details of the ISO standard, and the techniques of operational semantics. Unfortunately, such people are hard to find, which is rather an indictment of the divergence between theory and practice in computer science.

Fortunately, we can still use the theorem prover to test our definition by proving a number of minor, and obvious, results.[1] We expect these theorems to be true in advance, and we can be sure that they don't reflect any semantic profundity. However, if we can prove them it is a validation of our semantics. If, on the other hand, it turns out that what we are attempting to show is false, then our proof will quickly founder on a case that embodies this falsity. We will then realise that either our semantics or our original claim is incorrect. We must still decide this question at a level above the logic, but the process of the attempted proof has focussed our attention on a part of the semantics that might have otherwise escaped our attention.

We will now present a selection of invariants for our dynamic semantics. In almost all cases these are proved by performing a rule induction over the relevant subset of the semantics. As well as giving each a brief precis, we give the number of lines of SML that the proof required. This is a primitive measure of proof complexity, but will provide a rough means of comparing the complexities of these proofs with those that come later in this section.

- Undefined sub-expressions never disappear. (36 lines)

  $$\begin{array}{ll} & \mathcal{U} \text{ an } \textit{active} \text{ sub-expression of } e_0 \wedge \langle e_0, \sigma_0 \rangle \rightarrow_e \langle e, \sigma \rangle \\ \Rightarrow & \mathcal{U} \text{ an } \textit{active} \text{ sub-expression of } e \end{array}$$

  where an *active* sub-expression is one on the LHS of a sequence point inducing operator such as `&&` or the comma-operator.

  The original statement of this theorem did not have the requirement that the undefined sub-expressions be active, in the sense given above. This form of the statement is false however because an undefined sub-expression that was on the RHS of a `&&` operator could disappear if the LHS of that operator evaluated to zero.

- The initialisation map is always a subset of the allocation map. (30 lines)

  $$I_{\sigma_0} \subseteq \Lambda_{\sigma_0} \wedge \langle s, \sigma_0 \rangle \rightarrow \langle v, \sigma \rangle \Rightarrow I_\sigma \subseteq \Lambda_\sigma$$

  This result holds for all of the relations defined in the semantics, and all possible pieces of syntax.

- An expression is *lval_safe* if the RHS's of all assignment expressions within it are always either wrapped in a `RVR` construct, are values,

[1] Rushby calls these theorems *formal challenges* [Rus93].

are undefined, or are binary operators. This property is preserved by expression evaluation. (24 lines)

$$\mathit{lval_safe}(e_0) \wedge \langle e_0, \sigma_0 \rangle \rightarrow_e \langle e, \sigma \rangle \Rightarrow \mathit{lval_safe}(e)$$

This result is useful because it reassures us that we will never get lvalues at the top level of the RHS of an assignment expression. Such a situation would cause the semantics to block as the $\mathcal{L}$ context for allowing lvalue to value reduction doesn't cover this syntactic form.

- Having a finite bag of pending side effects is preserved by expression evaluation. (16 lines)

$$\texttt{FINITE_BAG}(\Pi_{\sigma_0}) \wedge \langle e_0, \sigma_0 \rangle \rightarrow_e \langle e, \sigma \rangle \Rightarrow \texttt{FINITE_BAG}(\Pi_{\sigma})$$

These results are a representative sample of those actually proved in the course of the Cholera mechanisation. We also have a similar set of trivial identities that serve the same purpose of validating our definition. Here are two of them:

- Sequential compositions involving the empty statement are equivalent to those without it. (19 lines)

$$\langle \texttt{;} :: \mathit{stmts}, \sigma_0 \rangle \rightarrow_s \langle v, \sigma \rangle \equiv \langle \mathit{stmts}, \sigma_0 \rangle \rightarrow_s \langle v, \sigma \rangle$$

- Trap distributes over `if` statements. (8 lines)

$$\begin{array}{rl} & \langle T(b, \texttt{if (}g\texttt{)}\ s_1\ \texttt{else}\ s_2), \sigma_0 \rangle \rightarrow_s \langle v, \sigma \rangle \\ \equiv & \langle \texttt{if (}g\texttt{)}\ T(b, s_1)\ \texttt{else}\ T(b, s_2), \sigma_0 \rangle \rightarrow_s \langle v, \sigma \rangle \end{array}$$

Of the invariant results given above, all but the first are what we term "syntactic preconditions": they embody properties that we want to hold when a program begins to execute. Reasoning about programs where we have infinite bags of pending side effects, for example, is a pointless exercise. These preconditions do not appear in the definition of the dynamic semantics but we are not interested in situations where they do not hold. Sometimes it is possible to prove results without such assumptions (the two identities above, for example), but more complicated theorems about C's behaviour are almost invariably accompanied by a series of assumptions stating that the values we are interested in satisfy various preconditions.

Now, preconditions of this form have a great deal in common with a type system seen in its rôle as a program filter that rejects programs that

are not well-formed. In this light, all of our syntactic preconditions are just extensions to the type system. On the other hand, one difference is that they apply not just to the syntax of the program, but also make requirements of the program state as well. The fact that these preconditions are invariant, that they are preserved by evaluation, is vital because we want them to hold not only when a program begins its execution but also at all intermediate stages as well.

The question that naturally arises is then whether or not the original typing relation is preserved by the evaluation relation, just as our "extensions" are preserved. This would be an important validation of the type system as it would demonstrate that the evolution of syntax (expressions in this case, given the absence of types for statements and declarations) proceeds in a constrained way. This property is known as *subject reduction*, as well as type preservation. It holds for our definition of C (though we require that the resulting expression be free of undefined sub-expressions, written $\mathcal{U}$-**free**(e)):

Theorem 1 (Subject reduction) *If $\langle e_0, \sigma_0 \rangle \rightarrow_e \langle e, \sigma \rangle$, and our initial expression is well typed (i.e., $\Gamma_{\sigma_0}, \Sigma_{\sigma_0} \vdash e_0 : \tau$) then as long as e is $\mathcal{U}$-**free**, e will have the same type as e_0. That is, we will have $\Gamma_\sigma, \Sigma_\sigma \vdash e : \tau$.*

This is proved in the mechanisation as an immediate consequence of a stronger theorem which states both the conclusion above, and that even if a sub-expression does become undefined, then all other sub-expressions in the whole will retain a type. This stronger result is useful in other work, but takes 230 lines of SML to prove. The result as stated (if proved on its own) requires 190 lines. The proof is a rule induction over all of the rules for expression reduction in the semantics, and is not particularly interesting.

Once stated and proved, the property that being well-typed is preserved is useful in subsequent proofs. It is of particular use in rule inductions over the dynamic semantics when considering cases corresponding to rules where static properties are assumed. An example of this is the rule for `structs`: it is only the guarantee of well-typedness that ensures that a reference to a member will succeed. In the determinism results to be described later in this chapter, we are often concerned to show that a reduction can be mimicked by another, analogous, reduction starting from a slightly different initial state. In these situations, we need to be sure that these "type-dependent" reductions will go ahead.

This frequent need to refer to an assumption of well-typedness is the price we pay for omitting static details in the rules for the dynamic semantics. On the other hand, the case for including the static semantics as part

of the dynamics applies equally well to all the other syntactic preconditions we have discovered, and including these in the dynamics would complicate an already overly complicated semantics. It is our claim that it is better to separate the various facets of the semantics so that they can be considered in isolation, even if this does complicate the statement of later results.

Being well-typed is *not* a guarantee that an expression will not give rise to undefined behaviour. Subject reduction allows us to conclude that if a well-typed expression reduces to a value, then this value will be of the same type as the original expression, but as we saw in chapter 2, the property of being defined is not syntactically checkable. Instead, we can prove a characterisation of the ways in which the reduction semantics can "block". (This takes 193 lines to prove.)

Theorem 2 (Typed expressions' blocking behaviour) *Given a state* σ_0*, if* e_0 *is well-typed (i.e.,* $\exists \tau.\, \Gamma_{\sigma_0}, \Sigma_{\sigma_0} \vdash e_0 : \tau$*) and lval_safe, but is not a value, is not* $\mathcal{U}$ *and does not include any function calls, then there exist an expression* e *and a state* σ *such that* $\langle \mathit{RVR}(e_0), \sigma_0 \rangle \rightarrow_e \langle \mathit{RVR}(e), \sigma \rangle$*.*

The requirement that the expression not include a function call highlights the fact that expressions *can* block if they include within them a function call that would loop if executed. The rule for function calls requires that there be a state reached by the statement execution; if there isn't one, this rule blocks. The conclusion says that we are guaranteed a reduction only if the expression is inside a RVR constructor because an lvalue at the top level will not reduce on its own. However, as previously noted, all expressions in the statement and declaration semantics are evaluated with such a wrapper, so our statement is not too specific to be useful. It we take our result's contrapositive, we gain a characterisation of the possible forms a typed expression will take when it does block.

4.2 Symbolic evaluation and its pitfalls

C's expressions are difficult to work with in the context of verification. Here we will briefly explain the nature of this difficulty, thereby motivating the work that is to be explained in the remainder of the chapter.

The problem with C's expressions is that they are non-deterministic. Non-determinism makes program verification difficult because one has to verify that all possible outcomes satisfy the chosen specification. Without any more precise knowledge as to how an expression will behave and what all these outcomes will be, a verification must examine each possible order of evaluation so that each possible outcome can be checked against

the specification's post-condition. Consider for example the simple binary expression `i * j++`. Here there are just four events:

1. evaluating `i`,
2. evaluating `j++`, yielding the value of `j` and putting the side effect of adding one to that value into the bag of pending side effects,
3. applying the side effect, and
4. performing the multiplication

Using the numbers above to identify the events, the following sequences of events are all possible: 1234, 2134, 1243, 2143, and 2314. In larger expressions with multiple side effects, the explosion of possibilities becomes even worse. And yet the expression given is actually deterministic. Subsequent sections in this chapter will look at how we can demonstrate determinism for two broad classes of expressions.

For the moment, we examine how we might trace out all of an expression's possible execution paths. (In the worst case, where we do not have a result to the effect that an expression is deterministic, it is this procedure to which we will have to resort, exponential blow-up or not.) As discussed in section 2.7, it is entirely straightforward to turn an inductive definition in the mechanisation into a set of rewrites that perform evaluation. As a result, Cholera can perform expression evaluation quite readily. Furthermore, one is not constrained to evaluate only expressions that are entirely ground terms, as one would be in an interpreter for the language. Instead, the operation of logical rewriting will introduce any disjunctions necessary if the behaviour of the program depends on the contents of memory or some other variable. This is thus a naïve form of *symbolic evaluation*. For example, given the term $\langle e_0, \sigma_0 \rangle \to_e \langle e, \sigma \rangle$ where e_0 is a concrete piece of syntax, this rewriting will generate a set of logical constraints on the variables in the term, removing $\to_e$ entirely.

If one is not interested in staying within the logic, a more sophisticated approach (albeit one that eschews the LCF guarantees of soundness) is available in Syme's DECLARE system [Sym97a]. Syme's approach is to take the logical specification of an operational semantics and use it to generate SML code for performing execution. This automatically generated interpreter can then be used as a convenient (and very satisfying) way of validating one's semantics on realistic program examples. This is possible because the compilation of the definition into SML provides a level of efficiency not possible within the logic.

Further, Syme's earlier work on mechanising SML [Sym93] (building in this case on work by Hutchins [Hut90]) demonstrated that it is possible to generate more efficient implementations of symbolic evaluation, even within the logic. However, this approach was not automated and would require considerable time to implement in the case of C, not so much because of the number of rules, but because of their complicated nature, particularly in their interactions with the program state. In any case, no matter how efficiently one implements symbolic evaluation, it is the nature of C that there will be exponentially many cases to consider whenever an expression includes binary operators.

Symbolic evaluation implemented by simple-minded rewriting does appear to be adequate for statement evaluation in the absence of loops, and that is the approach pursued in Cholera, as is further described in chapter 5.

4.3 Pure expressions

Our first attack on C's non-deterministic expressions proceeds by way of first concentrating on another "unwholesome" aspect of C's expressions, the fact that they may have side effects. Under the assumption that expressions which don't have side effects will prove more tractable, our approach is to develop a theory of such expressions, which we will call *pure*. This property is characterised at the semantic level thus:

$$\begin{array}{rcl} pure(e_0) & \hat{=} & \forall \sigma_0, \sigma, e.\, \langle e_0, \sigma_0[\Pi := \emptyset, R := \emptyset, \Upsilon := \emptyset]\rangle \to_e^* \langle e, \sigma\rangle \implies \\ & & \forall a.\, a \in \Lambda_\sigma \implies M_\sigma(a) = M_{\sigma_0}(a) \end{array}$$

(An expression is $pure$ if the contents of memory (modulo what is currently allocated) are always the same at all stages.)

Basic computability theory tells us that deciding whether or not an expression is $pure$ is impossible (Rice's theorem again). For this reason we develop a secondary notion of purity that is based entirely on the syntax of an expression. This will provide a simpler, decidable check for purity. We expect that syntactically pure expressions will be a subset of the $pure$ expressions, giving us a sound but incomplete characterisation of the semantic notion.

Our definition of what it is to be syntactically pure is straight-forward, if naïve: it rejects expressions containing function calls, assignment expressions, post-increments and post-decrements. A more sophisticated analysis would be less hasty in forbidding function calls and would allow function calls that were to functions that did not include impure expressions. This approach would require traversing functions' call-graphs (requiring some

sort of inductive definition) and we avoid this because it seems overly complex. We write $synpure(e)$ to mean that e is syntactically pure in this way.

We first prove that the property of syntactic purity is preserved by evaluation:

$$\forall e_0, \sigma_0, e, \sigma.\ \langle e_0, \sigma_0 \rangle \rightarrow_e^* \langle e, \sigma \rangle \wedge synpure(e_0) \implies synpure(e)$$

This proof is 19 lines of ML, and proceeds by rule induction over $\rightarrow_e$. We next prove that syntactically pure expressions are all semantically pure. In fact, we prove a slightly stronger result: that syntactically pure expressions change only the reference map component of a state. Abusing our notation somewhat, we write that a *state* is pure when it has an empty update map, and when it has no side effects pending. We further write $\sigma_1 \sim_R \sigma_2$, when σ_1 is identical to σ_2, except for the reference map component. Given this definition we have:

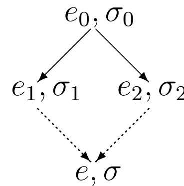
Lemma 3 (Syntactic purity implies semantic purity)

$$\forall e_0, \sigma_0, e, \sigma.\ \langle e_0, \sigma_0 \rangle \rightarrow_e^* \langle e, \sigma \rangle \wedge synpure(e_0) \wedge pure(\sigma_0) \implies \sigma \sim_R \sigma_0$$

This is again proved by rule induction, this time taking 18 lines. (Because $\sigma_1 \sim_R \sigma_2$ implies that all but reference map components are identical, we have met the conditions for an expression to be $pure$, as per the definition. We also have that σ is $pure$ in the overloaded sense given above.)

4.3.1 Proof outline

We should like to demonstrate determinism by showing a *diamond property* for all of the possible reductions that a syntactically pure expression might undergo. Graphically, this amounts to showing that in all situations we can find reductions to fill in the dashed lines below:



It follows that if this can be shown for single steps of a reduction relation, then that reduction system must be confluent. Unfortunately, this property

does not hold for pure expressions. A counter-example is `(x && y) + x`. If this expression evaluates the `x` on the RHS of the addition before the sequence point is reached, then the final state of the evaluation will not record `x` in its reference map. If however, the evaluation delays its evaluation of the RHS until after passing the sequence point introduced by the logical and, then this reference *will* be present in the final state's reference map. As a result, we will only be able to show determinism up to $\sim_R$.

Furthermore, the result also fails if there is a undefined expression as an immediate sub-expression of e_0. For example, if we have $\mathcal{U}$ `+ i`, then this can undergo two possible reductions: one to $\mathcal{U}$, and one to $\mathcal{U}$ `+` v, where v is the value of the variable `i`. It is not possible to reduce to anything from the first result, which is entirely undefined. However, expressions which contain undefined sub-expressions have already exhibited undefined behaviour, so there is little point in describing their behaviour. Therefore we add the precondition that e_0 not include any undefined sub-expressions. We will investigate the determinism of undefined behaviours in section 4.5.

These provisions notwithstanding, this is still a useful result as the reference map components of final states are always ignored in evaluating one expression to the next within statements. (The fact that all of a program's expressions are always evaluated in a state that has its reference map set to $\emptyset$ is easily confirmed by inspecting the rules for statements and declarations that include expressions within them.)

4.3.2 Pure expression determinism

We begin by proving the powerful result that our relation $\sim_R$ between states is preserved in a way that is reminiscent of bisimulation.

Theorem 4 ($\sim_R$ is a "pseudo-bisimulation" for $synpure$ expressions)

$$\begin{array}{l} \forall e_0, \sigma_0, e, \sigma.\ synpure(e) \wedge pure(\sigma_0) \wedge \langle e_0, \sigma_0 \rangle \rightarrow_e \langle e, \sigma \rangle \implies \\ \quad \forall \sigma_0'.\ \sigma_0 \sim_R \sigma_0' \implies \\ \quad\quad \exists \sigma'.\ \langle e_0, \sigma_0' \rangle \rightarrow_e \langle e, \sigma' \rangle \ \wedge \ \sigma \sim_R \sigma' \end{array}$$

This result is proved by a straightforward induction over the rules for $\rightarrow_e$, and is proved in HOL with 69 lines of ML. This result makes the actual diamond result quite easy to prove.

Theorem 5 (Single-step diamond result for $synpure$ expressions)

$$\begin{array}{l} \forall e_0, \sigma_0, e_1, \sigma_1, e_2, \sigma_2. \\ \quad \langle e_0, \sigma_0 \rangle \rightarrow_e \langle e_1, \sigma_1 \rangle \ \wedge\ \langle e_0, \sigma_0 \rangle \rightarrow_e \langle e_2, \sigma_2 \rangle \ \wedge\ (\exists \tau.\, \Gamma_{\sigma_0}, \Sigma_{\sigma_0} \vdash e_0 : \tau) \ \wedge \\ \quad pure(\sigma_0) \wedge synpure(e_0) \ \wedge\ \mathcal{U}\textbf{-free}(e_0) \ \wedge\ (e_1, \sigma_1) \neq (e_2, \sigma_2) \\ \quad \implies\ \exists e, \sigma_1', \sigma_2'. \\ \qquad\qquad \langle e_1, \sigma_1 \rangle \rightarrow_e \langle e, \sigma_1' \rangle \ \wedge\ \langle e_2, \sigma_2 \rangle \rightarrow_e \langle e, \sigma_2' \rangle \ \wedge\ \sigma_1' \sim_R \sigma_2' \ \wedge \\ \qquad\qquad (\mathcal{U}\textbf{-free}(e) = \mathcal{U}\textbf{-free}(e_1) \ \wedge\ \mathcal{U}\textbf{-free}(e_2)) \end{array}$$

We prove this by inducting over the structure of the expression e_0. To illustrate the proof, we will describe the four significant cases that arise for binary operators, focussing on the diamond part of the conclusion rather than the $\mathcal{U}$**-free** properties. Each case presents us with a situation in which there have been two reductions as per the hypothesis of the theorem above, i.e., two possible ways of achieving $\langle e_{01} \odot e_{02}, \sigma_0 \rangle \rightarrow_e \langle e, \sigma \rangle$ with two resulting value-state pairs $\langle e_1, \sigma_1 \rangle$ and $\langle e_2, \sigma_2 \rangle$.

1. The two reductions are both in the left hand argument of the operator. So we have $\langle e_{01}, \sigma_0 \rangle \rightarrow_e \langle e_1, \sigma_1 \rangle$ and $\langle e_{01}, \sigma_0 \rangle \rightarrow_e \langle e_2, \sigma_2 \rangle$, with the top level $\langle e_{01} \odot e_{02}, \sigma_0 \rangle$ reducing to either $\langle e_1 \odot e_{02}, \sigma_1 \rangle$ or $\langle e_2 \odot e_{02}, \sigma_2 \rangle$. This case is handled by our inductive hypothesis: we have that this dual movement in the left sub-expression implies two further reductions to a common result: that there exist e, σ_1', and σ_2' such that $\langle e_1, \sigma_1 \rangle \rightarrow_e \langle e, \sigma_1' \rangle$, $\langle e_2, \sigma_2 \rangle \rightarrow_e \langle e, \sigma_2' \rangle$, and $\sigma_1' \sim_R \sigma_2'$. The existence of these reductions implies reductions at the higher level as well, and thus we show the desired conclusion.

2. The two reductions are both in the right hand argument. This case is directly analogous to the first, and the argument is the same.

3. The two reductions are in e_1 and e_2. So we have $\langle e_{01}, \sigma_0 \rangle \rightarrow_e \langle e_1, \sigma_1 \rangle$ and $\langle e_{11}, \sigma_0 \rangle \rightarrow_e \langle e_2, \sigma_2 \rangle$. By lemma 3 we have that $\sigma_1 \sim_R \sigma_0 \sim_R \sigma_2$. This in turn implies by theorem 4 that we can make analogous reductions to the originals in the different starting states, that there exists σ_1' and σ_2' such that $\sigma_1 \sim_R \sigma_1'$, $\sigma_2 \sim_R \sigma_2'$, $\langle e_{01}, \sigma_2 \rangle \rightarrow_e \langle e_1, \sigma_1' \rangle$ and $\langle e_{02}, \sigma_1 \rangle \rightarrow_e \langle e_2, \sigma_2' \rangle$. At the level of the whole binary expression, we therefore have that $\langle e_1 \odot e_{02}, \sigma_1 \rangle \rightarrow_e \langle e_1 \odot e_2, \sigma_2' \rangle$ and $\langle e_{01} \odot e_2, \sigma_2 \rangle \rightarrow_e \langle e_1 \odot e_2, \sigma_1' \rangle$. Our finishing states are related by $\sim_R$, so we are done.

4. There are two reductions taking values to different results, i.e., from $\langle \underline{(v_1, \tau_1)} \odot \underline{(v_2, \tau_2)}, \sigma_0 \rangle$ to two different results. We show that this is a contradiction by appealing to the fact that the binary operators are in fact deterministic.

Our result is not quite a canonical diamond result. For the reasons already given, we can not expect that both of our reductions will be able to find a common state to reduce to, but we do at least have that they can reduce to two states which are almost exactly the same. Moreover, these fresh reductions do not introduce any undefined sub-expressions. This latter point means that if we insist on e_1 and e_2 being $\mathcal{U}$**-free**, then we can be sure that the resulting expression e will be as well. We will need to insist on this condition if we are to be able patch multiple diamonds together and draw conclusions about the behaviour of $\rightarrow_e^*$. Upon extending our result so that it applies to $\rightarrow_e^*$, we get the following "determinism" result.

Theorem 6 (Determinism of $synpure$ **expressions)**

$$
\begin{array}{ll}
\forall e_0, \sigma_0, v_1, \tau_1, \sigma_1, v_2, \tau_2, \sigma_2. & \\
& \langle e_0, \sigma_0 \rangle \rightarrow_e^* \langle \underline{(v_1, \tau_1)}, \sigma_1 \rangle \ \wedge \ \langle e_0, \sigma_0 \rangle \rightarrow_e^* \langle \underline{(v_2, \tau_2)}, \sigma_2 \rangle \ \wedge \\
& synpure(e_0) \ \wedge \ pure(\sigma_0) \\
\Longrightarrow & (v_1 = v_2) \wedge (\tau_1 = \tau_2) \wedge (\sigma_0 \sim_R \sigma_1 \sim_R \sigma_2)
\end{array}
$$

4.4 Sequence point free expressions

While pure expressions have many nice properties (in addition to being deterministic, they can be easily adapted for use in a Hoare style programming logic for statements, as we shall see in chapter 5), it is a fact of life that many expressions in idiomatic C programs are not pure. In this section, we present another class of expressions which are deterministic even though they are not necessarily pure. This class is that of expressions without internal sequence points. Recall that sequence points occur in the evaluation of the logical operators `&&`, `||` and `?:`, as well as in the comma-operator as well when a function call is made. If we eliminate these syntactic forms we obtain a different class of expressions, those which are *sequence point free*.

The sequence point free expressions overlap the pure expressions but neither set is included in the other. There are also a number of expressions which are neither pure nor sequence point free. At the end of this section we will suggest one further class of deterministic expressions, but even then none of our classes extend to cover expressions including function calls. If any set of expressions in C is non-deterministic it will be those expressions that include function calls. For example, our semantics for function calls makes the following program non-deterministic, but defined:

```
int x = 4;

int f(void) { x = 3; return x; }

int main(void) { return x + f(); }
```

Given our semantics, this program will return either 6 or 7 depending on whether the function call happens before or after the evaluation of the `x` in the expression in the `main` function. On the other hand, an alternative semantics which interleaved the execution of function bodies with execution of the surrounding expression would make this program undefined. Given the standardisation committee's seeming unwillingness to confront this issue, our semantics seems as plausible as any other.[2]

Recall the example presented earlier: `i * j++`. This is a sequence point free expression, and it is deterministic. An informal argument that this is the case would simply rest on the observation that the evaluation of `i` and the side effect incrementing `j` can't possibly interact. In expressions where an interaction is possible, such as in `i + i++`, we don't get non-determinism. Instead undefined behaviour is the result. This stems from the restrictions on the reference and update map components of the state (R_σ and Υ_σ). Two rules from chapter 3 are particularly relevant. The first is where we are attempting to read from memory, and must observe the restriction that the region of memory read (r below) is not in the update map.[3]

$$\frac{\Upsilon_\sigma \cap r = \emptyset \quad r \subseteq I_\sigma \quad \texttt{okVal}(\sigma, m, \tau) \quad \tau \text{ not an array type}}{\langle \texttt{LV}(a,\tau), \sigma\rangle \rightsquigarrow \langle \underline{(m,\tau)}, \sigma[R := R + r]\rangle}$$

$$\begin{aligned} r &= \{a \ldots a + |\tau| - 1\} \\ m &= M_\sigma\langle a \ldots a + |\tau| - 1\rangle \end{aligned}$$

The second rule is where updates are made, and where there is a similar restriction: new updates can not update memory already updated or referred to:

$$\frac{\eta \in \Pi_\sigma \quad s \subseteq \Lambda_\sigma \quad R_\sigma \cap s = \emptyset \quad \Upsilon_\sigma \cap s = \emptyset}{\langle e, \sigma\rangle \rightarrow_e \langle e, \sigma[I := I \cup s, M := M[r := m], \Pi := \Pi - \{\eta\}, \Upsilon := \Upsilon \cup s]\rangle}$$

[2]We admit our semantics' failings with respect to termination and non-determinism (brought about by the use of a big-step semantics at the level of statements). Nonetheless, fixing this and using a small-step semantics for statements will not in itself resolve the question of whether function calls interleave, as expressing both semantics is possible in this framework.

[3]The accompanying rule specifies that undefinedness results if this rule's preconditions are not met.

where
$$\begin{array}{rcl} \eta &=& \clubsuit(a, m) \\ r &=& \langle a \ldots a + |m| - 1 \rangle \\ s &=& \{a \ldots a + |m| - 1\} \end{array}$$

For brevity's sake we now define an auxiliary function, `apply_se`, which takes a side effect and a state and returns the state described above. In other words, the RHS of the above rule could be rewritten `apply_se`(σ, η). Our proof of determinism will follow the same outline as we used for pure expressions: showing confluence for successful evaluations, while deferring the treatment of undefined behaviour to section 4.5.

4.4.1 A sequence point free diamond property

Even assuming that our reduction sequences do not become undefined, the task of proving determinism for expression evaluation is quite complicated. In particular, the system as described is made difficult to reason about by the fact that side effect applications and other forms of reduction can intermingle. The first stage of our proof is to demonstrate that side effect applications can all be postponed to the end of an evaluation sequence without affecting the result.

This should be clear from the constraints described earlier: if a side effect application were to make a difference, a subsequent reference to memory would need to look at some part of memory that the side effect had changed; but this is precisely one of those situations forbidden (a reference to updated memory) and would lead to undefinedness, contradicting our earlier assumption.

The proof proceeds by first showing that side effect applications and other reductions can commute.

Lemma 7 (Pure reductions commute with `apply_se`) *For all expressions e_0, e_1, for all states σ, σ_0, σ_1, and for all side effects η, if η is pending in σ, with $\sigma_0 =$ `apply_se`(σ, η) (i.e., σ_0 is the state that results from applying η to σ), and $\langle e_0, \sigma_0 \rangle \rightarrow_e \langle e, \sigma_1 \rangle$ then there exists a state σ' such that $\langle e_0, \sigma \rangle \rightarrow_e \langle e, \sigma' \rangle$, η is pending in σ' and $\sigma_1 =$ `apply_se`(σ', η).*

This is a straightforward rule induction on the inductive definition of $\rightarrow_e$. Another induction readily extends this to allow side effect applications to be pushed past any number of other expression reduction steps. Using this, we then induct on the number of reductions to prove our "separation theorem":

Theorem 8 (Separation) *For all expressions e_0, e, and for all states σ_0, σ, if $\langle e_0, \sigma_0 \rangle \rightarrow_e^* \langle e, \sigma \rangle$, then there exists a state σ' and a sequence of side effects*

$\eta_1 \ldots \eta_n$ where both the update maps and memory contents of σ_0 and σ' are the same, and $\langle e_0, \sigma_0 \rangle \rightarrow_e^* \langle e, \sigma' \rangle$ and σ is the result of applying the side effects $\eta_1 \ldots \eta_n$ to σ'.

(Note that the final value e is present after the expression reduction steps, and before the side effect applications begin. This is because these later applications can not change the value that an expression yields.)

We now consider the $\rightarrow_e$ relation as the union of two components: reductions where no side effect applications occur, and reductions that are exclusively side effect applications. Let us use $\rightarrow_E$ for the former and $\rightarrow_A$ for the latter so that $\rightarrow_e = \rightarrow_E \cup \rightarrow_A$. Confluence for both $\rightarrow_E$ and $\rightarrow_A$, together with the separation theorem imply confluence for $\rightarrow_e$ as follows:

1. Consider two reduction sequences starting at $\langle e_0, \sigma_0 \rangle$ that both complete normally. One is to $\langle e_1, \sigma_1 \rangle$ and the other is to $\langle e_2, \sigma_2 \rangle$.

2. By the separation theorem, both reduction sequences can be separated into two phases, with intermediate points $\langle e_1, \sigma_1' \rangle$ and $\langle e_2, \sigma_2' \rangle$, such that $\langle e_0, \sigma_0 \rangle \rightarrow_E^* \langle e_1, \sigma_1' \rangle$ and $\langle e_1, \sigma_1' \rangle \rightarrow_A^* \langle e_1, \sigma_1 \rangle$ (similarly for e_2 etc.)

3. Because e_1 and e_2 represent completed evaluations, they must be values. As $\rightarrow_A$ only applies side effects, it doesn't change expressions. Thus the states reached by $\rightarrow_E^*$ must be terminal with respect to it. Then if $\rightarrow_E$ is confluent, these intermediate states are actually the same.

4. Now we have two reduction sequences involving $\rightarrow_A^*$ from the same starting point. As $\rightarrow_A$ is also confluent, the final states are necessarily identical.

Given this result, we need only prove that $\rightarrow_E$ and $\rightarrow_A$ are confluent.

Confluence for $\rightarrow_E$

We establish confluence for $\rightarrow_E$ by demonstrating a diamond property for single steps of the relation.

Before beginning a proof such as this, it is instructive to consider parallels with the similar task that one faces in attempting to prove confluence for the λ-calculus. There, things are somewhat complicated by the fact that a reduction in the RHS of a β-redex may have to be matched by many repetitions of essentially the same reduction in an alternative branch where the

RHS has been substituted into the body of the LHS. This doesn't happen in our semantics, where substitution doesn't arise.

However, the λ-calculus is at least entirely syntax-directed; if a redex is present, then the reduction can always take place, and its result will always be the same. Reductions in the λ-calculus can be said to ignore their context. This is not the case in the C semantics where the accompanying state, an ever-present and varying context, can affect reductions. This is not just a matter of different values for variables affecting the value of an expression, but more significant: a state with a large update map may make a reduction that would otherwise turn a variable into a value instead produce undefinedness.

With this motivation behind us, the first stage in our proof will be to characterise the degree to which states can vary and yet still produce the same reduction for a given piece of syntax. Furthermore, because expression reductions affect the state[4], we want to characterise the way in which this happens, so that, ultimately, we will be able to state that reduction x can reduce in the same way both before and after reduction y.

Theorem 9 (Reduction characterisation) *If $\langle e_0, \sigma_0 \rangle \rightarrow_E \langle e, \sigma \rangle$ then there exists a function f characterising the reduction, such that $f(\sigma_0) = \sigma$, and for all σ_0' which are "no more restrictive" than σ_0, then $\langle e_0, \sigma_0' \rangle \rightarrow_E \langle e, f(\sigma_0') \rangle$.*

The meaning of "no more restrictive" above turns out to be rather detailed in its expression, really suitable only for the consumption of a theorem prover.[5] In essence it requires that the update map be no bigger in σ_0' than it is in σ_0, but there are also are a number of conditions required of both the initial states and the expressions involved. One of these is that e_0 be well-typed. Another is that e not be undefined; computations that do allow e to become undefined are discussed in section 4.5.

We also have the following important lemma, which like the previous is established by induction over the reduction relation.

Lemma 10 (Reduction preconditions preserved) *If $\langle e_0, \sigma_0 \rangle \rightarrow_E \langle e, \sigma \rangle$, then σ is no more restrictive than σ_0 in the sense of theorem 9.*

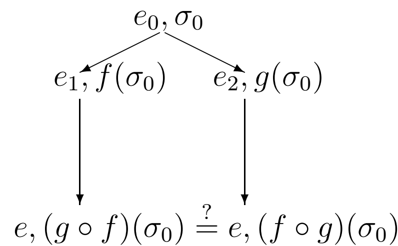
Now we can prove the diamond property for $\rightarrow_E$ relatively straightforwardedly. Again an induction is required over the reduction relation. The

[4]Though $\rightarrow_E$ holds update maps and thus memory constant, we will still get new side effects being added to the queue of those pending, and as objects are referred to, reference maps also increase.

[5]However, the full statements of this and our other theorems are available in Appendix B.

inductive or "sub-expression" cases, where we have two reductions within the same sub-expression, are handled by the inductive hypotheses, so it is just the cases where an expression form admits two reductions in different sub-expressions which prove difficult. This includes both the normal binary operators, and also assignment, which needs to be treated separately because unlike the other operators, it adds a side effect to those pending.

In such a situation, our reduction characterisation and reduction preconditions results tell us immediately that a "diamond" of four sides can be constructed. If the functions required to exist by the first result are f and g, then the diagram looks like:



The question then remains as to whether or not f and g will commute. They do in fact, as each does little more than specify the additions to the starting state's reference map and pending side effects. Addition on bags being commutative, the result follows.

Confluence for $\rightarrow_A$

The second requirement of the proof of is to show that the $\rightarrow_A$ relation is confluent. We show this by demonstrating a diamond property. This is a considerably simpler task than for $\rightarrow_E$.

Recall that we are performing reductions in a context where all of the side effects can be applied successfully, resulting in normal termination with a value. This implies that no pair of pending side effects affect overlapping parts of memory. We show this by contradiction. One of the side effects must have been applied first. Subsequent to this application, the other side effect can not have been applied because this would result in undefined behaviour (two updates of the same part of memory). But if the second side effect is not applied, then the final state must still have side effects pending, which also contradicts our assumption, because a normal termination is a sequence point, by which state all side effects must have been applied.

So, all of the side effects affect different parts of memory, and can therefore be applied independently of one another. The required diamond property is an immediate consequence of this.

Sequential expressions

A broader class of deterministic expressions is those that have their sequence point introducing operators at the top-level of the syntax tree. Such expressions are thus made up of a sequence of sequence point free expressions. An example of one such would be

```
(x && y = x + z, z = y ? 0 : z + 1, y + 10)
```

Given our earlier result, this expression is clearly deterministic, though a proof of this fact has not yet been accomplished in the Cholera mechanisation.

4.5 Undefined evaluations

In this section we aim to show that if an evaluation sequence exists which leads to undefined behaviour, this undefinedness can not be escaped, and that all states reachable from the initial one must necessarily either be undefined themselves, or still admit the possibility of becoming undefined in one or more steps. This result then makes it clear that a normal terminating evaluation and an undefined one can not both begin from the same initial state. We reason as follows: assume that such a situation exists. Then our undefinedness result states that it is possible to reach undefinedness from the final state of the normal evaluation. But if it is a final state, then it can not take any more steps, and it is not in an undefined state itself because it has yielded a proper value. Thus we have a contradiction and an assurance to the effect that all evaluations are in fact deterministic.

We begin by defining the notion of state *safety*. A state is *safe* if its pending side effects conflict neither with each other (i.e., do not affect overlapping parts of memory), nor with the state's reference map and update map. It should be clear that a state which is safe can apply all of its side effects without becoming undefined. The converse is the basis of our first lemma in this section. (This notion is not relevant in the $pure$ "world" as $pure$ states are necessarily safe by virtue of not ever having any pending side effects.)

Lemma 11 (Finite and unsafe states can become undefined) *If a state σ_0 is both unsafe and has a finite bag of pending side effects, then for all e_0 there exists a reduction sequence and a state σ such that $\langle e_0, \sigma_0 \rangle \rightarrow_e^* \langle \mathcal{U}, \sigma \rangle$.*

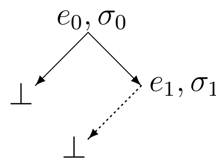
Also, for all e_0, e, σ_0 and σ, if σ_0 is unsafe, and $\langle e_0, \sigma_0 \rangle \rightarrow_e \langle e, \sigma \rangle$, then σ is also unsafe.

Our semantics represents undefinedness arising as a result of expression evaluation (e.g., division by zero, or a reference to a variable already updated) by replacing the offending expression with $\mathcal{U}$ in the syntax tree and then letting this "bubble" its way to the top of the tree. This can not be prevented:

Lemma 12 (Undefined sub-expressions can always ascend) *If an expression e_0 is not $\mathcal{U}$-***free***, then for all σ_0 there exists a reduction sequence and a state σ such that $\langle e_0, \sigma_0 \rangle \rightarrow_e^* \langle \mathcal{U}, \sigma \rangle$.*

These two results (neither of which is particularly surprising) make it clear that a large class of expression-state pairs, those which are unsafe or which have undefined subexpressions, though not necessarily "fully undefined", might as well be. We shall refer to such states as *effectively undefined*. Though a state's being effectively undefined may not seem such a strong claim initially, the condition preservation clauses of the lemmas above should make it clear that an effectively undefined state is one which can never yield a value. In conjunction with the fact that all sequence point free and $synpure$ expressions must terminate[6], we can see that effectively undefined means "will necessarily become undefined".

Our next theorem is more significant. We wish to show that if a reduction occurs which makes something effectively undefined, when it was not effectively undefined before, then if one takes a different step from the same initial state, the result will either be effectively undefined, or it will retain the ability to make a reduction to an effectively undefined state. This can be represented as a "broken" diamond, where we write $\perp$ for states which are effectively undefined in the sense defined earlier:



Another analogy is that of the cliff-edge. Over the edge lies effective undefinedness. Once one reaches the edge, one can walk along it, but while it may be possible to avoid falling over the edge for some indeterminate length of time, it is not possible to move away. For $synpure$ expressions, this property follows immediately from theorem 5. In the context of sequence point free expressions, the proof proceeds in a similar way to that of the proof of the confluence of $\rightarrow_E$.

[6] Neither class includes function applications.

While the inductive cases are straightforward, we need to cope with the fact that the reduction from $\langle e_0, \sigma_0 \rangle$ to $\langle e_1, \sigma_1 \rangle$ might involve a reduction in a sub-expression unrelated to that which produced the undefinedness. Inside $\langle e_1, \sigma_1 \rangle$ we want to have a reduction occur that is analogous to the one that produced undefinedness from $\langle e_0, \sigma_0 \rangle$. We do this by again establishing a reduction characterisation result, and by demonstrating that reductions preserve this.

In this case, the characterisation is essentially that an analogous reduction to undefinedness can occur in any state that is at least as restrictive as the original. This condition is preserved both by $\rightarrow_E$ and $\rightarrow_A$.

We then do an induction of the number of steps along the cliff's edge to produce:

Theorem 13 (The cliff's edge) *For all e_0, σ_0, if $\langle e_0, \sigma_0 \rangle \rightarrow_e \langle e_1, \sigma_1 \rangle$ and $\langle e_1, \sigma_1 \rangle$ is effectively undefined, then for all e_2 and σ_2 such that $\langle e_0, \sigma_0 \rangle \rightarrow_e^* \langle e_2, \sigma_2 \rangle$, there exists e' and σ' such that $\langle e_2, \sigma_2 \rangle \rightarrow_e \langle e', \sigma' \rangle$, and $\langle e', \sigma' \rangle$ is effectively undefined.*

In the context of sequence point free expressions, we still need to add one more diamond property. This is a surprisingly easy proof as it does not require an induction over the meaning relation. Instead the characterisation functions and our lemma 7 that $\rightarrow_E$ and $\rightarrow_A$ commute combine to give:

Theorem 14 (A diamond property for $\rightarrow_E$ and $\rightarrow_A$) *For all e_0, e_1, e_2, σ_0, σ_1, σ_2: if $\langle e_0, \sigma_0 \rangle \rightarrow_E \langle e_1, \sigma_1 \rangle$ and $\langle e_0, \sigma_0 \rangle \rightarrow_A \langle e_2, \sigma_2 \rangle$ and both $\langle e_1, \sigma_1 \rangle$ and $\langle e_2, \sigma_2 \rangle$ are not effectively undefined, then there exist e and σ (possibly effectively undefined), such that $\langle e_1, \sigma_1 \rangle \rightarrow_A \langle e, \sigma \rangle$ and $\langle e_2, \sigma_2 \rangle \rightarrow_E \langle e, \sigma \rangle$.*

The final proof is now possible. We wish to show that if a reduction sequence takes an initial state ($\langle e_0, \sigma_0 \rangle$) to undefinedness, then all other possible destinations from the same starting point retain this possibility. In essence, we exploit the possibility of completing a confluent diamond on the cliff-tops.

1. We have a reduction sequence from $\langle e_0, \sigma_0 \rangle$ to undefinedness. Let $\langle e_1, \sigma_1 \rangle$ be the last state in this sequence not effectively undefined.

2. We have another reduction sequence to $\langle e_2, \sigma_2 \rangle$, and by assumption this is not effectively undefined.

3. Therefore, either using all three of our diamond properties for $\rightarrow_E$ and $\rightarrow_A$, or using our determinism result for all pure expressions, we have a common possible destination for both $\langle e_1, \sigma_1 \rangle$ and $\langle e_2, \sigma_2 \rangle$. Call this $\langle e, \sigma \rangle$.

4. Having come along the cliff's edge from $\langle e_1, \sigma_1 \rangle$, $\langle e, \sigma \rangle$ must still be on the edge, thereby retaining the possibility of a reduction to an effectively undefined state, if it is not an effectively undefined state already.

5. Effectively undefined states all allow for a reduction sequence to "full" undefined-ness, so $\langle e_2, \sigma_2 \rangle$ must do so as well by virtue of being able to reduce to $\langle e, \sigma \rangle$.

Chapter 5

Statements

We describe work that allows the user to reason about C programs at the level of statements. We present the derivation of a restricted Floyd-Hoare logic for C programs, and also discuss the utility of this approach. We then extend this to include a system for analysing loop bodies and automatically generating provably correct post conditions, even in the presence of `break`, `continue` *and* `return` *statements.*

5.1 A programming logic for C

C's dynamic semantics is particularly complicated. Reasoning directly with it in the theorem prover is a frustrating and unpleasant business. At this level of work, proofs about concrete programs are conducted by performing a symbolic evaluation of the code. We have already discussed the combinatorial explosion that confronts one when the evaluation of expressions is performed. Here we discuss ways of simplifying the task of working at the level of statements.

As motivation, we follow the example set by Gordon [Gor89], where a programming logic of partial correctness is derived from a (denotational) semantics. The programming logic derived is one that resembles the axiomatic semantics presented in Hoare's seminal paper [Hoa69]. Our aim is to do the same, deriving a logic for reasoning about C's statements. For all that this logic may appear axiomatic, it is important to remember that it is not an independent statement of the semantics. It is instead a consequence of our operational definitions, and necessarily sound as a result. That this is a reasonable aim is borne out by the observation that C does not have any particularly complicated statement forms. Nonetheless there are two major problems to deal with: expressions have side effects, and some statement

forms allow for the flow of control to be interrupted.

The problem of side effects in expressions is a particularly difficult one to reconcile with a system along the lines suggested by Hoare. The main cause of difficulty is the fact that we should like to be able to equate expressions at the level of the programming language with expressions at the level of the predicates which assert facts about the state of the abstract machine. When expressions are necessarily side effect free, this is both reasonable and easy to specify. However, our coercion between levels becomes more complicated with the introduction of side effects. Before we can proceed with a consideration of the problems that our expression semantics introduces, we should also describe exactly what we mean by our "Hoare triples". Our definition is[1]

$$\{P\}\ s\ \{Q\} \mathrel{\hat{=}} \forall \sigma_0, \sigma, v.\ P(\sigma_0) \wedge \langle s, \sigma_0\rangle \rightarrow_s \langle v, \sigma\rangle \implies Q(\sigma) \wedge (v \neq \mathcal{U})$$

Now we are in a position to consider a rule where an expression needs to be lifted to the level of state-predicates. Hoare's original rule for the `if` statement is[2]

$$\frac{\{P \wedge B\}\ S_1\ \{Q\} \quad \{P \wedge \neg B\}\ S_2\ \{Q\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{Q\}}$$

Here B plays the rôle of both guard expression in the syntax of the statement, and of general predicate on states. Our own first attempt at a rule for C's `if` statement will not be so lax about typing, using an explicit *lift* function to inject our syntax into the type of predicates on states. Given our definition of *lift* it is also important that we write *lift*$(\texttt{!}g)$ (where `!` is C's logical negation operator) rather than $\neg$*lift*(g).

$$\frac{\{P \wedge \mathit{lift}(g)\}\ s_1\ \{Q\} \quad \{P \wedge \mathit{lift}(\texttt{!}g)\}\ s_2\ \{Q\}}{\{P\}\ \texttt{if (}g\texttt{)}\ s_1\ \texttt{else}\ s_2\ \{Q\}}$$

where $\mathit{lift}(e)(\sigma) = \exists m, \tau, \sigma'.\ \langle \texttt{RVR}(e), \sigma\rangle \rightarrow_e^* \langle \underline{(m, \tau)}, \sigma'\rangle \wedge V_\tau(m) \neq 0$

Unfortunately, the presence of side effects makes this rule incorrect. In particular, when in the context of the `if` statement, the statements s_1 and s_2 will not be evaluated in the same contexts as that assumed by the antecedents of the rule. Consider the case where the guard of the `if` statement is the expression `i++`. This expression will cause the execution of the

[1] We have simplified slightly; we actually require that our triples have a fourth parameter which corresponds to the environmental components of the state that both σ_0 and σ will share. Expecting code to be valid with respect to a specification given all possible environments is so tough a requirement as to be meaningless.

[2] We update Hoare's original syntax of $P\{s\}Q$ to the now preferred $\{P\}\ s\ \{Q\}$.

second arm of the conditional whenever `i` is zero, but that statement will be not be executed in such an environment, because the act of evaluating the guard will make `i` non-zero.

Since Hoare's paper in 1969, there has naturally been a lot of work investigating this problem. For example, Boehm [Boe85] suggests that a form of dynamic logic can be used. This work relies on expressions being both deterministic and terminating, properties that do not hold of C expressions in general, but is otherwise a very appealing approach as it also handles aliasing elegantly. An implementation of this work in Cholera would require an embedding of Boehm's first order logic so that substitution over the logic's terms could be defined. It would also need to address the issues of non-determinism and non-termination, probably by using the rules only for those expressions already known to be deterministic and terminating.

More recently, Black has developed a limited form of axiomatic semantics for C [BW96, BW98], where side effects are factored out of expressions by translating statements where they occur to semantically equivalent formulations that do not have the side effects in the significant expressions. This work does not prove the correctness of its semantic transformations with respect to any underlying semantics, and is foundationally shaky as a result. Rather, it focuses on the use of HOL as the basis for reasoning on top of its axioms. Proving Black's rules correct with respect to Cholera's operational semantics would both be a validation of his work, and would provide a precise characterisation of those forms for which his approach is valid. However, this is not such an appealing prospect because of the need to use semantic equivalence.

An alternative approach is presented by Kowaltowski [Kow77]. There the suggestion is to wrap expressions in pre- and post-conditions as well, and to explicitly deal with the value of the expression as part of the post-condition. This approach has the merit of being straightforward but it does not cope with the sort of non-determinism possible in C programs. In particular, it is predicated on the assumption that individual sub-expressions of a whole expression will be totally evaluated before other expressions are considered. C's expressions are not so well-behaved, and there does not seem to be an obvious way of extending Hoare triples down to expressions from the level of statements.

All of the above suggests that this is indeed a difficult area. However, we must not lose sight of the eventual aim, which is to provide a simpler way of reasoning about C programs. Our logic's completeness is of less importance than its utility in conjunction with whatever other tools we may be able to develop. Moreover, there is no point in developing an "axiomatic" semantics if it turns into a restatement of the original operational seman-

tics. Instead, we use our other tools to handle expressions. When they are known to be deterministic for example, we can evaluate them with just one evaluation strategy, eliminating the combinatorial explosion of possibilities.

This is not to say that there is no rôle for our putative programming logic. While our assumption of a useful symbolic evaluation tool for expressions would seem to simplify most statement forms, we still have to consider loops. These can not be evaluated symbolically, and having an invariant rule for them would obviate the need to perform a rule induction each time one was encountered. Here then is our first result, which we state for O, the general loop construct that underlies all three of C's loop forms:

Theorem 15 (The C invariant rule)

$$\frac{\{I\}\ \texttt{if}\ (g)\ s\ \texttt{else}\ ;\ \{I\}}{\{I\}\ O(g,s)\ \{I\}}$$

Our proof is quite simple (in the theorem prover the following is just 20 lines of SML tactic):

Assuming that I is invariant over the `if` statement, that I holds of σ_0, and that there is a reduction of the form $\langle O(g,s),\sigma_0\rangle \rightarrow_s \langle v,\sigma\rangle$, we want to show that $v \neq \mathcal{U}$ and $I(\sigma)$.

We argue by rule induction over all statement reductions. There are four rules for loops (given in section 3.4.5):

1. *The guard g evaluates to $\mathcal{U}$. But this is impossible given that we have $\{I\}$ `if` (g) s `else ;` $\{I\}$, and $I(\sigma_0)$, as our definition of the Hoare triple requires that the result value be defined. So this is a contradiction.*

2. *The loop guard evaluates to false, and the loop exits directly. Given our invariant I holds for both possible execution paths of the `if` statement, it will hold after the execution path consisting of executing the guard, getting a false result, and executing the empty statement. If I holds of the state after the empty statement is executed, then it will hold of the state before it. This state corresponds to the state immediately after the loop exits, so we have $I(\sigma)$, as required.*

3. *The loop guard evaluates to true, and the loop body is entered. However, it does not exit normally so the loop exits abnormally too. The same execution path of "guard returns true, body exits abnormally" is part of the possible behaviour of the `if` statement, so again the result follows.*

4. *Finally, the inductive case. From the text of the rule, we have:*

$$\frac{\langle \mathit{RVR}(g), \sigma_0[\Pi := \emptyset, R := \emptyset, \Upsilon := \emptyset]\rangle \rightarrow_e^* \langle \underline{(m,\tau)}, \sigma_1\rangle}{\langle s, \sigma_1\rangle \rightarrow_s \langle \mathtt{StmtVal}, \sigma_2\rangle \qquad \langle O(g,s), \sigma_2\rangle \rightarrow_s \langle v, \sigma\rangle}$$

The guard succeeds, the body executes normally once, and the loop is entered once more. We appeal to the inductive hypothesis: to show that $I(\sigma)$ holds, we show that it holds of σ_2 (and also that the environmental components of the state at σ_2 are the same as they were at σ_0). But again, σ_2 is the end-point of a possible execution of the `if` *statement, so the result follows.* □

We can specialise this result to the `while` statement, by first proving the following result about the trap (T) form.

$$\frac{\{P\}\ s\ \{Q\}}{\{P\}\ T(v,s)\ \{Q\}}$$

As the `while` statement just traps `continue` and `break` interruption results at the appropriate levels, the next result is immediate:

$$\frac{\{I\}\ \mathtt{if}\ (g)\ s\ \mathtt{else}\ ;\ \{I\}}{\{I\}\ \mathtt{while}\ (g)\ S\ \{I\}}$$

We do not attempt to specialise our result so that it holds of the other two concrete loop forms (the `do-while` and `for` loops). Instead, we expect the propagation of the verifier's invariant into the various complexities of these forms' definitions to be handled by our symbolic evaluation facilities.

Our invariant rules have the advantage of bypassing both problems already identified with C statements, that of side effects in expressions, and the various statement interruption forms. This is because the `if` statement includes both components present in the loop, and if something is already invariant over the side-effecting guard and the possibly interrupted body of the conditional statement, then it will retain these properties when part of a loop. Our rule finesses these issues back to a level where symbolic evaluation is possible.

For example this result may be useful when analysing code such as the following, which searches a linked list (on the assumption that the first element (pointed to by the pointer variable `uptr`) is not the one we want to find):

```
while ((uptr = uptr->link) != NULL) {
  if (search_condition(uptr))
    return uptr;
}
```

This code has both a side effect in its guard, and a break to the flow of control in the middle of the loop (the `return` statement). Our rule tells us that we can demonstrate an invariant property for the loop by demonstrating one for the `if` statement instead.[3] However, our rule has at least one serious flaw not present in Hoare's formulation: we do not learn anything new about the state of the program if the loop terminates. His rule is

$$\frac{\{I \wedge B\}\ S\ \{I\}}{\{I\}\ \textbf{while}\ B\ \textbf{do}\ S\ \{I \wedge \neg B\}}$$

We can't come to a similar conclusion in C, even with a pure guard, because of the possibility that the loop will exit due to the action of an interruption statement (`break` or `return` in particular; `continue`, though it does interrupt blocks, doesn't cause any of the standard loop forms to exit). We will discuss a comprehensive solution to this problem in the next section, but in its absence, the best we can do is the following:

$$\frac{s\ \text{contains no}\ \texttt{break}\ \text{or}\ \texttt{return}\ \text{statements}}{\{P\}\ O(g, s)\ \{\textit{was}(\texttt{!}g)\}}$$

The *was* predicate takes a C expression and a state and is true if the state is one which is the possible result of evaluating the expression and having that expression evaluate to true. This is a simple-minded definition, but it is another technique that allows one to bypass the issue of side effects. If the linked list example above were altered so that it did not have its `return` statement, this result would tell us that if it terminates, the loop will do so in a state where an assignment of the null pointer value has just been made to the `uptr` variable.

Our *was* function is just *lift* "in the other direction" and the standard HOL rewriting techniques that form the basis for symbolic evaluation will work just as well where the existentially quantified state variable is the starting state rather than the finishing state. If e is a pure expression, we have $\textit{was}(e) = \textit{lift}(e)$.

There is one final use to which we can put a development of logical rules for C's statements. Though access to symbolic evaluation will mean that we do not use them as part of a verification effort, they will characterise the degree to which C can be seen as a "better behaved" programming language. In particular, we will prove rules that are as similar to Hoare's originals as possible. However, because of C's various failings we will need to attach various side conditions to the rules we derive. If we ensure that these rules

[3] Our case study experience with the `strcpy` program in chapter 6 suggests that finding an appropriate invariant for the `if` statement may not be easy, or even possible.

are syntactic in nature (as with the syntactic characterisation of pure expressions in section 4.3), then we will have a syntactic characterisation of how C might be constrained to let simple axiomatic style reasoning work over it. In this spirit, we present the following rules:

$$\frac{\{P\}\ s_1\ \{Q_0\} \quad \{Q_0\}\ s_2\ \{Q\}}{\{P\}\ s_1\,;\ s_2\ \{Q\}}$$

where s_1 contains no interruption statements, and

$$\frac{\{P \land \mathit{lift}(g)\}\ s_1\ \{Q\} \quad \{P \land \mathit{lift}(\texttt{!}g)\}\ s_2\ \{Q\} \quad g \text{ pure}}{\{P\}\ \texttt{if (}g\texttt{)}\ s_1\ \texttt{else}\ s_2\ \{Q\}}$$

Unfortunately, there is no useful rule for assignment expressions (even those that appear on their own and not nested inside other expressions) in this setting. The best definition of assignment would be a recapitulation of the operational semantics, and of no real explicative power. The overly stringent nature of the restriction on the rule for sequencing would be addressed by a rule system that used multiple post-conditions, something that we will discuss in the next section.

Our final result allows us to describe when a loop will terminate. So far we have only been discussing *partial correctness*, where we are concerned to establish post-conditions if a statement terminates, but not whether or not that statement does actually terminate. Though Hoare's original paper does not provide rules for termination, it is well-known that one method is to demonstrate that a well-founded relation holds between all possible starting and finishing states of the loop body.

Our result is:

$$\begin{array}{ll} & (\forall \sigma_0.\ \mathit{terminates}\,(\texttt{if (}g\texttt{)}\ s\ \texttt{else ;})\,\sigma_0) \land \\ & (\exists f : \texttt{CState} \to \mathbb{N}.\ \forall \sigma_0\,\sigma.\ \langle s, \sigma_0\rangle \to_s \langle \texttt{StmtVal}, \sigma\rangle \land f(\sigma) < f(\sigma_0)) \\ \Longrightarrow & \forall \sigma.\ \mathit{terminates}\,(\texttt{while (}g\texttt{)}\ s)\,\sigma \end{array}$$

where $\mathit{terminates}\, s\, \sigma = \exists \sigma'\, v.\ \langle s, \sigma\rangle \to_s \langle v, \sigma'\rangle \land (v \neq \mathcal{U})$

The requirement that the `if` statement also terminates covers the possibility where the guard or body of the loop themselves go into infinite loops.

5.2 Automatic loop exit analysis

There are three statement forms that allow the flow of control to be interrupted in our definition of C: `break`, `continue` and `return`. All of these have the property that they cause flow of control to jump to a higher-level

part of the syntax within which they appear. By contrast, the goto statement, which we have omitted, allows the flow of control to jump down the abstract syntax tree. This property makes both reasoning about and defining the semantics of goto difficult, but the others are considerably more tractable. Note that while return statements can appear anywhere, break and continue statements can appear only in loop bodies.[4]

We have already seen how the presence of interruption statements limits the utility of our proof rule for loop statements. Where other rules are not so compromised by the lack of a good rule because of the possibility of general symbolic execution, rules about loops need to say as much as possible if we are to avoid forcing the user to perform some sort of induction. We would like to do the induction once, and then have the verifier exploit a simple derived rule that embodied that induction.

An example of a C program where such an analysis might be useful is given in figure 5.1. This program trims white-space from the end of a string and is adapted from an example in Kernighan and Ritchie [KR88]. In this example, it is not the case that the loop is sure to terminate with $n < 0$, as one might expect from an inspection of the guard. (Recall that C's for loop has the guard of the loop as the second expression between the parentheses.) Instead, this loop will also exit if the search backwards from the end of the string finds a non-space character. Idiomatic programs such as this one are the motivation for the development of the following theory.

```
int trim(char s[])
{
  int n;
  for (n = strlen(s) - 1; n >= 0; n--)
    if (!isspace(s[n])) break;
  s[n+1] = '\0';
  return n;
}
```

Figure 5.1: A program illustrating the use of break

A standard approach to the problem of interruption statements is to use a system of multiple post-conditions. This approach has an early descrip-

[4]The break and continue statements cause flow of control to return to the level of the *innermost* enclosing loop, while a return statement can be nested arbitrarily deep inside a function and still lift flow of control to the function's exit-point.

tion in the context of "structured" flow-chart programs with `goto` statements by Arbib and Alagić [AA79]. A more recent example is King and Morgan's work [KM95] in the setting of the refinement calculus. Each post-condition is associated with a different way in which execution may terminate, and each must hold if the statement in question terminates in the given way. King and Morgan only consider one form of interruption statement, which they call `exit`.[5]

Translating King and Morgan's presentation of a wp-semantics in the style of Dijkstra [Dij76] to one of Hoare-style "triples", we get the following rules (the $[\![\ ]\!]$ brackets are used to trap abnormal executions in the same way as T in our C semantics):

$$\frac{}{\{P\}\ \mathbf{skip}\ \{P\}\{\bot\}} \qquad \frac{}{\{P\}\ \mathbf{exit}\ \{\bot\}\{P\}}$$

$$\frac{\{P\}\ s\ \{Q\}\{Q\}}{\{P\}\ [\![s]\!]\ \{Q\}\{\bot\}} \qquad \frac{\{I \wedge B\}\ S\ \{I\}\{Q\}}{\{I\}\ \mathbf{while}\ B\ \mathbf{do}\ S\ \{I \wedge \neg B\}\{Q\}}$$

$$\frac{\{P\}\ s_1\ \{Q_1'\}\{Q_2\} \quad \{Q_1'\}\ s_2\ \{Q_1\}\{Q_2\}}{\{P\}\ s_1\, ;\ s_2\ \{Q_1\}\{Q_2\}}$$

$$\frac{\{P \wedge B\}\ S_1\ \{Q_1\}\{Q_2\} \quad \{P \wedge \neg B\}\ S_2\ \{Q_1\}\{Q_2\}}{\{P\}\ \mathbf{if}\ B\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \{Q_1\}\{Q_2\}}$$

The standard post-condition strengthening rule applies to both post-conditions. This is quite an elegant formulation of the semantics, but a translation to C would require four post-conditions (one for normal termination, and one for each of the other interruption statements), and this would be rather more tedious, both to define, and to work with.

The multiple post-conditions of the general form $\{P\}\ S\ \{Q_1\}\{Q_2\}$ are an abbreviation of the disjunctive claim, "if S exits (after starting in a state where P held), then it will do so normally, and Q_1 will hold, or it will exit abnormally, and Q_2 will hold." In our setting, the framework of the operational semantics already allows us to directly express how a statement has exited: given a reduction $\langle s, \sigma_0\rangle \rightarrow_s \langle v, \sigma\rangle$, we need only examine the value v. Therefore we achieve the expressiveness of the multiple post-condition programming logic by developing a system that analyses loop bodies and returns the appropriate disjunctions of possible post-conditions.

This has a deal in common with work on verification condition generation. Homeier [HM94], for example, has a system which constructs correct verification conditions for programs written in his simple language Sunrise.

[5] They also refer to earlier (unpublished) work by Back and Karttunen of Helsinki [BK83], which presents a general wp-semantics for statements with multiple exits.

Our aim is slightly different from that of the typical verification condition generator however. We wish to analyse a loop and derive a disjunctive description of all the conditions that might apply if and when the loop exits. When a loop doesn't contain any interruption statements, we are happy to simply return *was*$(\texttt{!}g)$, where g is the guard of the loop. We do not try to return loop invariants because we already provide a mechanism for demonstrating these using the derived rule of the previous section.

If a loop contains an interruption statement, we want to perform an analysis that will tell us the nature of the context in which that interruption might have occurred. If the loop contains a statement such as `if` (bc) `break;` then it is obvious that the context for the `break` statement is one in which bc has just evaluated to true. However, our contexts may not always be as simple as this. Consider extending the example so that it is:

$$\texttt{if } (bc)\ \{s_1 \ldots s_n\ \texttt{break;}\}$$

The statements $s_1 \ldots s_n$ might quite plausibly alter the state so that bc is no longer true. So, we need to return the information that if the loop exits due to a `break` statement, it may be in a state which is reachable by evaluating bc, having it return a non-zero value, and then executing all of the statements $s_1 \ldots s_n$. But what of the situation where the `if` statement in question is itself nested inside another `if` statement? To be maximally informative, we should continue to look further backwards to determine how the `break` statement came to be reached. In fact, to be complete, it is clear that we must look all the way back to the beginning of the loop.

The required loop analysis is done by the findint function, defined in figure 5.2. It can be used to either look for `break` or `return` statements, depending on its first parameter. The second, *events* parameter is a list of all the "events" that the search has passed over so far. The third parameter is the state that the loop terminates in. The final parameter is the syntax being searched. There are three different sorts of event; statement executions (denoted by *Stmt*), expressions evaluated as guards (denoted by *TGuard* and *FGuard* depending on whether or not the guard evaluates to true or to false), and variable declarations (denoted by *VDs*).

The function traverses the given syntax looking for the required form of interruption syntax. It returns a condition that is true if there is some way of reaching an interruption statement of the required form. It does this by accumulating a list of all the events that must intervene between the start of the execution at the head of the statement, and its eventual arrival at the statement form in question. If there is a such a sub-statement, it calls the translate function to calculate the conditions under which such an execution

$$
\begin{array}{l}
\mathsf{findint}(_,_,_,[e]\texttt{;}) = \bot \\
\mathsf{findint}(_,_,_,\texttt{continue;}) = \bot \\
\mathsf{findint}(rb, \textit{events}, \sigma, \texttt{break;}) = \\
\quad (rb = \texttt{break}) \wedge \exists\sigma_0.\ \mathsf{translate}(\textit{events} \frown \langle \textit{Stmt}(\texttt{break})\rangle, \sigma_0, \sigma) \\
\mathsf{findint}(rb, \textit{events}, \sigma, \texttt{return}\ e\texttt{;}) = \\
\quad (rb = \texttt{return}) \wedge \exists\sigma_0.\ \mathsf{translate}(\textit{events} \frown \langle \textit{Stmt}(\texttt{return}\ e\texttt{;})\rangle, \sigma_0, \sigma) \\
\mathsf{findint}(rb, \textit{events}, \sigma, \texttt{if (}g\texttt{)}\ s_1\ \texttt{else}\ s_2) = \\
\quad \mathsf{findint}(rb, \textit{events} \frown \langle \textit{TGuard}(g)\rangle, \sigma, s_1) \vee \\
\quad \mathsf{findint}(rb, \textit{events} \frown \langle \textit{FGuard}(g)\rangle, \sigma, s_2) \\
\mathsf{findint}(rb, \textit{events}, \sigma, O(g,s)) = \\
\quad \exists n \in \mathbb{N}. \\
\qquad \mathsf{findint}(rb, \textit{events} \frown \langle \textit{TGuard}(g); \textit{Stmt}(s)\rangle^n \frown \langle \textit{TGuard}(g)\rangle, \sigma, s) \\
\mathsf{findint}(rb, \textit{events}, \sigma, \{\textit{decls stmts}\}) = \\
\quad \mathsf{findintl}(rb, \textit{events} \frown \langle \textit{VDs}(\textit{decls})\rangle, \sigma, \textit{stmts})
\end{array}
$$

$$
\begin{array}{l}
\mathsf{findintl}(_,_,_,\langle\rangle) = \bot \\
\mathsf{findintl}(rb, \textit{events}, \sigma, s :: \textit{stmts}) = \\
\quad \mathsf{findint}(rb, \textit{events}, \sigma, s) \vee \\
\quad (s \notin \{\texttt{break}, \texttt{return}, \texttt{continue}\} \wedge \mathsf{findintl}(rb, \textit{events} \frown \langle \textit{Stmt}(s)\rangle, \sigma, \textit{stmts}))
\end{array}
$$

Figure 5.2: definition of the findint function

will end abnormally in state σ. This translate function takes a list of events and chains them together using the appropriate reduction relation. For example,

$$
\begin{array}{l}
\mathsf{translate}(\langle \mathit{TGuard}(g);\ \mathit{Stmt}(s_1);\ \mathit{Stmt}(s_2)\rangle, \sigma_0, \sigma) = \\
\quad \exists \sigma_1, \sigma_2, v. \\
\qquad \langle \mathtt{RVR}(g), \sigma_0[R := \emptyset, \Upsilon := \emptyset]\rangle \rightarrow_e \langle \underline{(m,\tau)}, \sigma_1\rangle \ \wedge V_\tau(m) \neq 0 \wedge \Pi_{\sigma_1} = \emptyset \ \wedge \\
\qquad \langle s_1, \sigma_1\rangle \rightarrow_s \langle \mathtt{StmtVal}, \sigma_2\rangle \ \wedge \\
\qquad \langle s_2, \sigma_2\rangle \rightarrow_s \langle v, \sigma\rangle \wedge v \in \{\mathtt{BreakVal}, \mathtt{RetVal}\}
\end{array}
$$

The rules for translate when it applies to variable declaration events (marked by the *VDs* form), are rather complicated and we omit the full details. Part of the reason for the complication stems from the fact that we are "flattening" a structural semantics into something that is linear and without structure. It is important to recall that in all cases, translate only holds if the chain of events ends with an abnormal execution.

Our limited form of condition generation differs from what is normally done (as typified by Homeier's work [HM94]) in two ways. Firstly, our conditions are generated directly in the higher-order logic. This is possible because we have no need to express the idea of substitution. This makes the task considerably simpler. The second difference is more significant: we express the generated condition by referring back to the semantics' reduction relations! Usually the point of verification condition generation is to remove such dependencies and finish with a set of conditions that are independent of the particular program. With findint, all we need is a valid post-condition; removal of dependencies on the reduction relations can come later with symbolic evaluation. Our principal result is:

Theorem 16 (Correctness of findint **function)** *If we have a reduction for a while loop such that* $\langle \mathtt{while}\ (g)\ s, \sigma_0\rangle \rightarrow_s \langle v, \sigma\rangle$*, then one of the following four possibilities must hold:*

1. $v = \mathcal{U}$*;*
2. $v = \mathtt{StmtVal} \wedge \mathsf{findint}(\mathtt{break}, \langle \mathit{TGuard}(g)\rangle, \sigma, s)$*;*
3. $v = \mathtt{StmtVal} \wedge \exists \sigma'. \langle \mathit{RVR}(g), \sigma'\rangle \rightarrow_e \langle \underline{(m,\tau)}, \sigma\rangle \wedge \Pi_\sigma = \emptyset \wedge V_\tau(m) = 0$*; or*
4. $\exists rv. (v = \mathtt{RetVal}(rv)) \wedge \mathsf{findint}(\mathtt{return}, \langle \mathit{TGuard}(g)\rangle, \sigma, s)$

The importance of this result is that it provides us with an automatic way of generating the disjunctive post-condition of a loop. Because findint performs a simple pass over the syntax of the loop's body, it is easily applied

in the HOL mechanisation. The only complication arises when a loop includes another loop as part of its body. In this situation, we can either apply our theorem a second time in order to characterise the termination state of the inner loop's execution, or we can simply ignore the inner loop and its predecessors in the list of events. After all, we are not required to trace a loop's history back as far as possible. The aim is simply to learn useful information about the loop's final state by knowing something about what preceded it, and there will always come a point where tracing a loop's execution backwards will not yield any more useful knowledge.

While the definition of the findint function is simple enough, the true test comes in attempting to prove that it does in fact generate correct post-conditions for loop execution. The proof of our theorem is embodied in some 900 lines of SML, and we won't attempt to replay all of the details of the proof here. However, the following is an outline of the lemmas needed to carry the proof through. First, we establish the equivalence of findint and translate. This proceeds by equating various translations of statements with translations of pieces of those statements in terms of event lists. We have, for example, that

$$\begin{aligned}
&\mathsf{translate}(\mathit{events} \frown \langle \mathit{Stmt}(\texttt{while (}g\texttt{) }s)\rangle, \sigma_0, \sigma) = \\
&\quad \exists n.(n \neq 0) \wedge \mathsf{translate}(\mathit{events} \frown \langle \mathit{TGuard}(g);\ \mathit{Stmt}(s)\rangle^n, \sigma_0, \sigma)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{translate}(\mathit{events} \frown \langle \mathit{Stmt}(\texttt{if (}g\texttt{) }s_1\texttt{ else }s_2)\rangle, \sigma_0, \sigma) = \\
&\quad \mathsf{translate}(\mathit{events} \frown \langle \mathit{TGuard}(g);\ \mathit{Stmt}(s_1)\rangle, \sigma_0, \sigma) \vee \\
&\quad \mathsf{translate}(\mathit{events} \frown \langle \mathit{FGuard}(g);\ \mathit{Stmt}(s_2)\rangle, \sigma_0, \sigma)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{translate}(\mathit{events} \frown \langle \{\mathit{decls}\ \mathit{stmts}\}\rangle, \sigma_0, \sigma) = \\
&\quad \exists p.\quad p \text{ is a prefix of } \mathit{stmts} \wedge \\
&\qquad\quad \mathsf{translate}(\mathit{events} \frown \langle \mathit{VDs}(\mathit{decls})\rangle \frown \mathbf{map}\ \mathit{Stmt}\, p, \sigma_0, \sigma)
\end{aligned}$$

All of these results are proved by performing an induction on *events* and then using standard equalities to equate the larger statement with the combination of fragments at the base case. The base case of the `while` loop needs two extra inductions (one for $\Rightarrow$, and one for $\Leftarrow$). With these equivalences in place, the similarities between translate and findint become clearer. These results also emphasise that it is translate which is really doing all of the work, giving us a means of pulling apart statements into smaller parts that "sum" to the whole. Once we have equivalences of the above sort for all possible statement forms, the final two equivalences, between the reduction relation $\rightarrow_s$ and translate, and between findint and translate follow quite readily. Our next result is

$$\mathsf{findint}(v, \mathit{events}, \sigma, s) = \exists \sigma_0.\, \mathsf{translate}(\mathit{events} \frown \langle \mathit{Stmt}(s)\rangle, \sigma_0, \sigma) \qquad (*)$$

We also have an analogous result for the accompanying findintl function:

$$\begin{array}{l} \mathsf{findintl}(v, \mathit{events}, \sigma, \mathit{stmtl}) = \\ \qquad \exists p, \sigma_0.\, p \text{ is a prefix of } \mathit{stmtl} \wedge \mathsf{translate}(\mathit{events} \frown p, \sigma_0, \sigma)) \end{array}$$

These equations follow quite readily (although one of the directions still requires 64 lines of tactic to prove!), and this is because we have the equations above for translate, which mimic the way in which findint works. Finally, we do have an easy result to the effect that

$$\langle s, \sigma_0 \rangle \rightarrow_s \langle v, \sigma \rangle \wedge (v \in \{\mathtt{break}, \mathtt{return}\}) = \mathsf{translate}(\langle \mathit{Stmt}(s) \rangle, \sigma_0, \sigma)$$

When we substitute a `while` loop for the s and rewrite with the set of equations above, the loop expands out into some non-zero number of repetitions of the loop guard and body. We can't tell what this number is, but as there was at least one repetition we have that

$$\begin{array}{rl} & \langle \mathtt{while}\ (g)\ s, \sigma_0 \rangle \rightarrow_s \langle v, \sigma \rangle \wedge (v \in \{\mathtt{break}, \mathtt{return}\}) \\ \Longrightarrow & \exists \sigma_0'.\, \mathsf{translate}(\langle \mathit{TGuard}(g);\ \mathit{Stmt}(s) \rangle, \sigma_0', \sigma) \end{array}$$

But the conclusion above is the same as the RHS of theorem $(*)$ above, and so we have the link to the two cases of our main result where findint is called. The other two cases of the main result specify the other possible results of evaluating the loop: that undefinedness may result (as always), as may a normal exit, caused by the loop's guard being false.

Chapter 6

Verification

It was always the aim of this research to investigate the ways in which our model of C, along with the accompanying "meta-level" theorems such as our determinism results, could be used to verify C programs. Having developed the model to a degree where we were confident in its correctness, and having also proved a number of results that promised to eliminate some of the obvious difficulties posed by C's semantics, we undertook an ambitious verification example. This example was a simple BDD implementation, but quickly proved to be too difficult. The reasons for this failure are examined in more depth in section 6.3. Though we were soon convinced that our BDD example was intractable, the experience of attempting it did serve to point out ways in which our basic tools might be improved. It even uncovered a few relatively minor faults in the underlying model.

After developing our verification technology further, we felt that the BDD verification was still likely to be impossible in the time available, and we instead chose to look at much simpler examples. These examples served to test the tools that we had developed, and their scale was in more keeping with other verifications done by researchers working with very detailed semantics. In contrast, Black's verification work [BW96, BW98] which assumes, but does not provide, a detailed semantic backing for very powerful techniques, is an indication of a future target for our own work. Sections 6.1 and 6.2 describe the two examples we succeeded in verifying. Finally, in section 6.4 we provide a description of two important tools that were significant in enabling the verification work that we performed.

6.1 Factorial

Our first example is the simple factorial program given in figure 6.1. While the factorial program is not a particularly exciting verification example (it is practically the canonical "toy" example), it was one done by Cook and Subramanian in [SC96], so we feel that there is some comparative value in pursuing it.

```
unsigned long fact(unsigned long n)
{
  unsigned long result = 1;
  while (n > 1) {
    result *= n;
    n = n - 1;
  }
  return result;
}
```

Figure 6.1: A program to compute factorials

The statement of correctness for this function is given in figure 6.2. The variables v_1 and v_2 and the constraints upon them express the idea that the state contains enough spare memory to allow for the allocation of two more variables of type `unsigned long` (the parameter `n` and the local variable `result`). (The $|$`long`$|$ expression represents the number of bytes required to store values of type `long` and `unsigned long`.) The only other initialisation constraint is that the initialisation map in σ_0 be a subset of the allocation map. The other necessary invariants required of the program state, such as those discussed at the beginning of chapter 4, are to do with expression evaluation, and all of the expressions evaluated in the course of the call to function `fact` automatically satisfy those constraints by virtue of being correctly initialised as part of statement evaluation.

Our statement of factorial's correctness differs from that in Subramanian and Cook by using the under-specified `ULONG_MAX` constant as the limit. Their verification of factorial is explicitly only for values of n up to and including 12, which gives the largest value of $n!$ representable with just 32 bits. Our model also forces us to consider memory allocation explicitly, so we have to add extra assumptions to allow our proof to go through. This requirement to model memory allocation initially seems to be extra work for no reward as there is no strictly conforming way for a C program to

$$
\begin{array}{l}
\forall \sigma_0\, n\, m_0\, v_1\, v_2. \\
\quad (\Phi_{\sigma_0}(\texttt{"fact"}) = \langle \textit{Code from figure 6.1} \rangle) \wedge n! \leq \texttt{ULONG_MAX} \wedge \\
\quad (V_{(\texttt{unsigned long})}(m_0) = n) \wedge I_{\sigma_0} \subseteq \Lambda_{\sigma_0} \wedge \\
\quad \texttt{alloc}(\sigma_0, |\texttt{long}|, v_1) \wedge \texttt{alloc}(\sigma_0, |\texttt{long}|, v_2) \wedge \\
\quad \{v_2 \ldots v_2 + |\texttt{long}| - 1\} \cap \{v_1 \ldots v_1 + |\texttt{long}| - 1\} = \emptyset \implies \\
\quad \exists \sigma\, m. \\
\qquad (V_{(\texttt{unsigned long})}(m) = n!) \wedge \\
\qquad \langle \widehat{\texttt{"fact"}}(\underline{(m_0, \texttt{unsigned long})}), \sigma_0 \rangle \rightarrow_e^* \langle \underline{(m, \texttt{unsigned long})}, \sigma \rangle
\end{array}
$$

Figure 6.2: Statement of the factorial program's correctness

cope with a failure to allocate automatic memory.[1] Given this, one's immediate reaction might be to call for a model which didn't require this seemingly pointless analysis of whether or not there is enough memory available. However, we believe such a model would be difficult to construct. More importantly, we believe that the requirement to explicitly account for a function's memory usage is likely to be a useful tool in the development of verified code, forcing verifiers and code developers to be aware of the demands their code will be making on what is ultimately a finite resource.[2]

The invariant used to verify this program was

$$n_0! = \texttt{result} * n!$$

(Where n_0 is the original value of the parameter `n`, and n is the current value.) We also showed that the loop was sure to terminate by demonstrating that the value of the variable n decreased with each iteration of the loop. This is all quite mundane.

Finally, note that the nature of the statement proved about the factorial program is only that there exists a final state satisfying the specification. On its own, this is not a logical guarantee that other paths of execution might not exist where the program exhibits different behaviours. Though we did not prove the corresponding universal result, we believe it to be true, given the determinism of all the components of the program. The only difference is that the universal result requires a stronger statement about the memory

[1] C's *automatic* variables are those such as locals and parameters that an implementation is likely to allocate on a run-time stack. By way of contrast, the `malloc` library call allows programs to request memory on the heap, but if the call fails, it returns a null pointer, making this an eventuality that the program(mer) can detect and deal with.

[2] All the virtual memory in the world can not enable a C program to use more memory than that addressable using its fixed-size, finite pointers.

that is available for allocation. In particular, the universal statement will require that there be available in unallocated memory either one block of $3 * |\texttt{long}| - 1$ bytes, or two non-contiguous blocks of $|\texttt{long}|$ bytes separated by at least one allocated byte. This is because we can not be sure that the model will make the best choices when it comes to allocate the first variable. With any fewer bytes available, it is not possible to guarantee that a particularly silly choice of allocation won't preclude allocating the next variable, thereby causing undefinedness. This solution of the "allocation problem" is used in the `strcpy` verification below, and also arose in our attempt to verify a BDD implementation, which is discussed in section 6.3.

6.2 `strcpy`

Our next example is taken from Kernighan and Ritchie [KR88], and is reproduced in figure 6.3. This is another short program, no doubt cryptically so for those with little C experience. This program is intended to copy strings, copying the second into the first. Strings in C are arrays of characters terminated by null bytes, and we know that arrays are handled by passing pointers to first elements, explaining the pointer parameters to the function. Its main loop does not have a body and consists of just a guard, but this is a guard that causes three side effects! (An expression of the form `*ptr++` returns the value stored at the address in `ptr` but also increments `ptr` so that it now points to the next value in memory.)

```
void strcpy(char *s, char *t)
{
  while (*s++ = *t++)
    ;
}
```

Figure 6.3: `strcpy`: a program to copy strings

Our statement of the correctness of this program is presented in figure 6.4. By way of contrast with the earlier factorial example, we here state a much stronger specification, completely characterising the resulting state in all possible evaluations. In the statement, the logical variables s_addr and t_addr correspond to the values of the function parameters, those which will be the initial values of the program variables `s` and `t` respectively. The precondition of the correctness statement includes three conditions on the sets

of addresses $\{\mathsf{t_addr} \ldots \mathsf{t_addr} + |t|\}$ and $\{\mathsf{s_addr} \ldots \mathsf{s_addr} + |t|\}$. These require that the destination string and the source string not overlap, that the source string be initialised, and that the destination string be part of allocated memory. Apart from the standard preconditions requiring that the initial state be *safe* and have its initialisation map be a subset of its allocation map, the only other requirement is that there be enough memory to allocate the two copies of the parameters passed into the function.

$$
\begin{array}{l}
\forall \sigma_0\ \mathsf{s_addr}\ \mathsf{t_addr}\ r\ \sigma.\\
\quad (\Phi_{\sigma_0}(\texttt{"strcpy"}) = \langle \textit{Code from figure 6.3} \rangle) \wedge I_{\sigma_0} \subseteq \Lambda_{\sigma_0} \wedge \textit{safe}(\sigma_0) \wedge \\
\quad (\exists v.\ \texttt{alloc}(\sigma_0, 3 * \texttt{sizeof}(\texttt{char *}) - 1, v)) \wedge \\
\quad \{\mathsf{s_addr} \ldots \mathsf{s_addr} + |t|\} \subseteq \Lambda_{\sigma_0} \wedge \{\mathsf{t_addr} \ldots \mathsf{t_addr} + |t|\} \subseteq I_{\sigma_0} \wedge \\
\quad \{\mathsf{t_addr} \ldots \mathsf{t_addr} + |t|\} \cap \{\mathsf{s_addr} \ldots \mathsf{s_addr} + |t|\} = \emptyset \wedge \\
\quad \mathsf{s_addr} \ldots \mathsf{s_addr} + |t| \text{ all representable addresses} \wedge \\
\quad \mathsf{t_addr} \ldots \mathsf{t_addr} + |t| \text{ all representable addresses} \wedge \\
\quad \langle \widehat{\texttt{"strcpy"}}(\underline{(\mathsf{s_addr}, \texttt{char *})}, \underline{(\mathsf{t_addr}, \texttt{char *})}), \sigma_0 \rangle \rightarrow_e \langle r, \sigma \rangle \implies \\
\quad (r = \underline{(\emptyset, \texttt{void})}) \wedge (I_\sigma = I_{\sigma_0} \cup \{\mathsf{s_addr} \ldots \mathsf{s_addr} + |t|\}) \wedge \\
\quad (\forall a \in I_\sigma.\ (M_\sigma(a) = M_{\sigma_0}[\langle \mathsf{s_addr} \ldots \rangle := t](a))) \\
\qquad\qquad\qquad\qquad\qquad \vee \\
\quad \exists \eta \in \Pi_{\sigma_0}.\ \sigma = \texttt{apply_se}(\sigma_0, \eta)
\end{array}
$$

where there is a null byte somewhere in memory after $\mathsf{t_addr}$*, thereby defining a finite sequence of characters, referred to above as* t*.*

Figure 6.4: Statement of the `strcpy` program's correctness

The conclusion of the correctness statement characterises the two possible outcomes of a single step of the expression reduction relation ($\rightarrow_e$) from a function call syntax-tree that has all of its arguments entirely evaluated. Either there is a side effect pending in the bag of pending side effects, and this is applied (the second possibility in the figure), or the function body is entered and the code is executed. This latter is the interesting case. By specifying that r must be a `void` "value" corresponding to a successful return from the function, our characterisation precludes the possibility of undefined behaviour having occurred.

We also characterise the initialisation and memory maps of the resulting state σ. There is no need to specify the other state components as an existing result states that they will be the same as in σ_0. We have completely characterised the initialisation map, but we can't completely specify the

contents of memory. In particular, we can't know where the function call chose to place its local parameters. In our model, where the memory map is a total function, those parts of memory will still be present. However, no program can reference memory that is outside of its initialisation map, so we completely specify the value of just that memory which is within the initialisation map.

Verifying `strcpy`'s correctness proved more difficult than expected. In particular, the variant of the Hoare rule for `while` loops presented in section 5.1 was not on its own adequate for dealing with the side effects in the loop. This is because it requires us to come up with an invariant for an `if` statement of the form

```
if (*s++ = *t++) ; else ;
```

In particular, our invariant must express bounds conditions for the variables `s` and `t` so that we can be sure that they do not access memory beyond the range of their respective arrays. Yet, because each execution of the given `if` statement is sure to increase both `s` and `t`, there is no possible invariant which imposes a finite upper limit on either variable.

Our solution to this problem was to demonstrate the equivalence of the given loop with the following:

```
while (*t) {
  *s = *t;
  s++;
  t++;
}
*s = *t;
s++;
t++;
```

This proof within a proof was done by inducting on the difference between the address of the null byte terminating the string t, and the address stored in `t`. With a side-effect free guard on its loop, this program is then amenable to the rule from section 5.1, and the proof of correctness can proceed relatively easily. Our invariant is given in figure 6.5, where it holds of state σ, and where state σ_0 and the address values s_addr and t_addr are as before. All but the last two conjuncts of the invariant establish required properties of the values of the two pointers `s` and `t`. The last two conjuncts of the invariant characterise the state's memory and initialisation maps respectively.

Apart from having to do an equivalence proof before the rest of the proof was possible, the major difficulty in this verification was discharging

$$\begin{array}{l}
\lambda\sigma.\ \exists tv\ sv. \\
\quad V_{\texttt{(char *)}}(M_\sigma\langle A_\sigma(\texttt{t}) \ldots A_\sigma(\texttt{t}) + \texttt{sizeof(char *)} - 1\rangle) = tv \wedge \\
\quad V_{\texttt{(char *)}}(M_\sigma\langle A_\sigma(\texttt{s}) \ldots A_\sigma(\texttt{s}) + \texttt{sizeof(char *)} - 1\rangle) = sv \wedge \\
\quad \mathsf{s_addr} \leq sv \wedge \mathsf{t_addr} \leq tv \wedge (tv - \mathsf{t_addr} = sv - \mathsf{s_addr}) \wedge \\
\quad tv \leq \text{address of } t\text{'s terminating null byte} \wedge \\
\quad (\forall a \in \Lambda_{\sigma_0}. \\
\quad\quad M_\sigma(a) = M_{\sigma_0}[\langle \mathsf{s_addr} \ldots \rangle := M_{\sigma_0}\langle \mathsf{t_addr} \ldots tv - \mathsf{t_addr}\rangle](a)) \wedge \\
\quad (I_\sigma = \\
\quad\quad I_{\sigma_0} \cup \{A_\sigma(\texttt{s}) \ldots A_\sigma(\texttt{s}) + \texttt{sizeof(char *)} - 1\} \cup \\
\quad\quad \{A_\sigma(\texttt{t}) \ldots A_\sigma(\texttt{t}) + \texttt{sizeof(char *)} - 1\} \cup \\
\quad\quad \{\mathsf{s_addr} \ldots sv - \mathsf{s_addr}\})
\end{array}$$

Figure 6.5: Invariant for the proof of `strcpy`'s correctness

side-conditions to do with the disjointness of ranges of addresses. These arise every time the contents of memory are examined, as we need to be sure that an update to some variable v_1 has not interfered with a previous update to another variable v_2. Though aliasing is particularly easy to establish in C, these problems are not unique to C. In this case, the presence of arrays would cause the problem to surface even in languages where aliasing between normal variables was not an issue.

Unsurprisingly, problems involving ranges of addresses required the use of HOL's arithmetic library. Unfortunately, this decision procedure performs poorly where arithmetic expressions contain many subtractions. As can be seen, the statements of our theorems use subtraction quite often, and this leads to slow and rather frustrating theorem-proving. It is possible that the theorems could be re-stated to reduce the number of occurrences of subtraction, but this would be at the cost of some clarity. It might be more appropriate to use HOL's recently acquired ability to exploit external decision procedures to solve these problems.

The `strcpy` example may seem to call into question the utility of the Hoare-like rule for `while` loops proved in section 5.1. It is certainly true that this rule did not solve the problem of the side effects in the guard. Instead, following Black's example in [BW96], we had to first transform our loop to a semantically equivalent form that had a guard free of side effects. While this was in itself an important part of the effort, we still had to prove that our invariant held over the new loop. This later stage of the verification proof *did* use our derived rule for `while` loops.

Another treatment of (a slightly different version of) `strcpy` is available

in Yuan Yu's thesis [Yu93]. Yu's approach was to verify the object code generated by a standard C compiler for the Motorola 68020 processor. This approach has the advantage of not needing to cope with any of C's under-specified wrinkles. The GNU C compiler used to produce the object code and the 68020 processor together eliminate the under-determinism present in the C semantics and result in a system that is one C implementation from the huge space of possible implementations. For example, this particular implementation does not trap accesses to uninitialised memory, so the specification of the problem in Yu's formulation is conceptually simpler. On the other hand, his specification of program behaviour has to specify the contents of processor registers and other similar low-level details. Yu verified more functions than just `strcpy` in his thesis, and although the `strcpy` assembly code, at only 13 lines, is not much worse than the C code, other examples demonstrate a big blow-up in code size when translated from C to assembly.

6.3 The failed BDD example

In wanting to demonstrate the applicability of the ideas of software verification in general and the Cholera system in particular, we pursued the goal of verifying a binary decision diagram implementation (see Bryant [Bry92] for a description of BDDs) by John Harrison [Har95a]. This choice had the advantage of being both interesting for a theorem-proving audience, and having a challenging, but seemingly not overly complicated, implementation. Further, the code to be verified did not require a model of system calls or any other features not modelled by Cholera.

We did a little work pursuing this goal, but it rapidly became apparent that this particular example was beyond the capabilities of the verification technology we had developed. The code in question was just over 400 lines long, and took roughly a weekend to implement. Our putative verification would have taken much longer.

We made the perhaps unwarranted assumption that the various semantic operations being performed (at a low level, list traversal; at a higher level, finding a key in a hash-table; at a higher level still, conjoining two BDD graphs) would be easy enough to reason about in the theorem-prover once they had been divested of their C-specific cladding. Unfortunately, we never had a chance to test this assumption as the task of dealing with the C embodiment of these abstract functions proved too difficult. In the remainder of this section, we will discuss some of the significant difficulties.

While conceptually simplest, one of the most significant pragmatic diffi-

culties was reasoning about large numbers. This problem arises because of the way in which HOL uses a unary representation (chains of applications of SUC to zero) to represent numerals. When attempting to verify a program that maintains its own internal heap of 10000 locations and a hash-table with 256 buckets, the various disjointness conditions generated involved many occurrences of these large numbers, which slowed down many aspects of theorem-proving. The latest version of HOL is about to improve this situation by adopting a binary representation for its numerals, but this improvement comes too late for this work.

The problem with numerals could be finessed to some extent by replacing them with under-specified constants. However, this would have exacerbated the other significant arithmetic problem. This was brought about by the fact that the BDD code declared two `struct` types. Subsequently, when arrays of these types were declared, the result was `sizeof` expressions involving multiplied sums. Dealing with multiplications involving large constants was difficult, but replacing the numeric constants with under-specified constants would have resulted in expressions beyond the capabilities of HOL's linear decision procedure to handle in general. Clever use of AC-normalisation and automatic expansion using the distributive law could handle many expressions, but having to manipulate the remaining expressions was tedious in the extreme.

Another problem we faced was brought about by a poor choice of technique in dealing with the verification of multiple functions in an environment with global variables present. We characterised the logical environment in which our functions executed by specifying it was the state which resulted from the declaration of the relevant global variables (including some `struct` declarations as well). Because variable declaration is nondeterministic in terms of the addresses chosen for globals, we used the Hilbert choice operator to pick one valid state. We thus had an expression along these lines:

$$\varepsilon\sigma.\ \langle \text{global declarations}, \sigma_0\rangle \rightarrow_v \langle \texttt{VarDeclVal}, \sigma\rangle$$

This was very painful to work with, but this problem was one that could have been avoided. Instead of using the above expression at all, we should have stated preconditions by characterising the various facets of the global environment independently. We might then have simply required that the initial state of the functions we were to verify had certain variable and type maps.

One last problem was the code's use of unportable tricks. For example, in the interests of efficiency, the code assumed that pointer values were

isomorphic to the even members of the `unsigned long` type, thus allowing the use of the low bit of the address's value as a long for storing extra information (in this case, whether or not the BDD link was negated). Ideally, we would have liked to state this assumption as a precondition to the verification. However, the Cholera mechanisation's handling of the representation of values is not ideal, and expressing this assumption would have been quite difficult. We eventually decided to try to verify a cleaned-up version of the code where no assumptions were made, and where the code was therefore strictly conforming. It is this verification that then went on to cause us all of the other difficulties explained above.

6.4 Verification tools

Our work on verifying code examples required the development of a number of verification tools. Here we describe two of the most significant. The first aimed to exploit the determinism result for sequence-point free expressions of section 4.4.1. There were two possible approaches to this task. We already had a theorem of the form

$$\langle e, \sigma_0\rangle \rightarrow_e^* \langle v_1, \sigma_1\rangle \wedge \langle e, \sigma_0\rangle \rightarrow_e^* \langle v_2, \sigma_2\rangle \implies (v_1 = v_2) \wedge (\sigma_1 = \sigma_2)$$

If, in the context of a particular proof, we also had a fact of the form $\langle e, \sigma_0\rangle \rightarrow_e^* \langle v, \sigma\rangle$, then together with the above theorem, an equivalence could be automatically derived stating that

$$\forall \sigma'\, v'.\ \langle e, \sigma_0\rangle \rightarrow_e^* \langle v', \sigma'\rangle \equiv (v' = v) \wedge (\sigma' = \sigma)$$

To be most useful, this approach required us to be able to rapidly generate a fact about a possible evolution of an expression to a value from a given starting point (an expression-state pair). This fact could then be the basis for the generation of a bespoke equivalence, as above. Unfortunately, these facts are not easy to generate. Simple rewriting will not work as this will produce big disjunctions corresponding to all of the possible non-deterministic paths. It is precisely these disjunctions that we are hoping to avoid. For example, a simple-minded rewrite starting with `i + j` will generate the following four top-level possibilities:

1. `j` evaluates one step to e', so the overall result is `i +` e'
2. `i` evaluates one step to e', so the overall result is e' `+ j`
3. there is a side effect pending, which is evaluated producing a different state, but keeping our initial expression unchanged

4. there are side effects pending, but none can be applied safely, resulting in undefinedness

The point of our proposed strategy is to choose one of these logical possibilities and create what would be in effect a derivation tree over the rules of the inductive definition of $\rightarrow_e$. However, implementing this strategy in HOL would require the writing of a fairly substantial amount of SML to make the right choices among the logical possibilities, which would simultaneously require performing the appropriate reasoning to discard choices that would lead to falsehood. (In our example above, picking either of the latter two choices will not make sense in a context where there are no side effects pending.)

Instead, our approach was to define a new constant in the logic which implemented a subset of the $\rightarrow_e$ relation. We then proved that the transitive closure of this new relation was equivalent to transitive closure of the original when the expression was sequence-point free and the result was a value. This new relation's defining equation could then be used as a simple HOL rewrite, safe in the knowledge that applying it would not cause a big case explosion. To deal with the possibility of evaluation to undefinedness, we could use our new relation to demonstrate an evaluation to a value from a sequence-point free starting point, and then invoke the theorems that stated that this precluded the possibility of undefinedness (theorem 13 and others in section 4.5).

This tool handled the evaluation of all of the expressions present in the two successful examples (including the nasty `*s++ = *t++`), and made verification of them tractable. We did not prove that our deterministic relation was also equivalent to the evaluation of pure expressions, which we would have needed to do to cope with some of the expressions in the BDD example. However, it is clear that this result follows, given the results from section 4.3.

Our second example of a verification tool is an automatic tool for deriving correct type judgements for (possibly partially evaluated) C expressions. The typing relation in turn depends on the well-formedness relation for types, so our tool has a similar automatic tool for that relation as a sub-component. Despite being obviously deterministic, both of these relations do not suit solution by simple rewriting. Instead our code for both the well-formedness and typing relations examines the input parameters and calculates a series of forward inferences that produces a theorem and, in the case of the typing relation, this theorem contains information as to what type the input expression has.

This technique, allowing the efficient derivation of special purpose theorems, is the same as that previously mentioned as having been used by Syme and Hutchins in their derivation of symbolic evaluators for SML [Sym93, Hut90]. It would also be the technique of choice for the symbolic evaluation of both our original $\rightarrow_e$ relation and its deterministic equivalent. Given an inductive definition of a relation, constructing the skeleton of the code to perform special purpose evaluation is relatively straightforward.

Assume that one is constructing an SML function that, when given a term x, is to return a theorem of the form $\vdash R(x, y)$. In the case of the expression typing relation, y is the type of the expression, while in the type well-formedness example there is no y parameter. For all rules of the form

$$\frac{R(x_1, y_1) \ldots R(x_n, y_n)}{R(x, y)}\ S$$

where S is some set of side conditions, presumably also relating the (x_i, y_i) pairs to x and y in the conclusion, and where the x in the rule matches the x we are interested in, we make recursive calls for the hypotheses above the line to solve for the $x_1 \ldots x_n$. (Our relations tend to be syntactically driven so knowing what the values of $x_1 \ldots x_n$ should be is usually straightforward.) Because there may be multiple matches in the rule set for any given x, our function should return a (lazy) list of possible solutions rather than just succeeding once or failing.

This much can be done quite automatically. The real work comes in discharging the side conditions S. These will not fit into any particular pattern, and so require a bit of thought from the tool developer. The expression typing and type well-formedness relations have minimal side conditions so weren't too difficult to implement. (They are also both simple enough to allow us to avoid having to use functions to lists of possible results.) Both the $\rightarrow_e$ relation and its deterministic partner have numerous side conditions, and we felt that simple rewriting, though slow, would suffice.

The expression typing relation tool was used a great deal in the course of the verification proofs that we performed. An expression is often required to have a type as a theorem precondition, so the application of this tool enabled theorems to be applied more easily.

Our two successful case study verifications (factorial and `strcpy`) were not very complicated. However, they are of similar complexity to programs verified by others working with detailed semantics. Their successful completion helped verify our underlying model, led to the development of new tools, and in conjunction with the attempt to verify the BDD code, pointed the way ahead to future, larger-scale verifications.

Chapter 7

Conclusion

The underlying thesis of this work is that formal methods can be fruitfully applied to real world programming languages. The foundation of our work is the definition of C's static and dynamic semantics presented in chapters two and three. In formally describing more of the language than has been managed in the past, this represents a step forward in our knowledge about the C language. Our definition also avoids mistakes made by others. We further believe that our definition has been carefully constructed. By explicitly setting out to use the standard [ISO90] as our first reference, we aimed to avoid the interpretive errors possible when using other sources. We also made a great deal of use of the `comp.std.c` Usenet newsgroup, a forum which provides ongoing debate about the standard's various intricacies. By following (and occasionally participating in) the newsgroup's various debates, one's knowledge of the standard can only improve.

In formalising a significant portion of the ISO standard, we have learnt a great deal about C's various nooks and crannies, but a formal definition in itself is of marginal utility. On the one hand, a lone definition such as ours is difficult to accept as correct with respect to the original natural language definition. Our own experience in finding problems with earlier definitions suggests that it is very easy to make mistakes in this area. Further, even if we assume correctness, we should like to demonstrate that a formal definition leads to useful consequences. For example, it is often held that the precision of formal language definitions helps in the development of compilers. The first version of the ML Kit compiler for SML [BRTT93] is a very good example of this, having been developed such that "for every inference rule in the formal semantics, there is is corresponding piece of code in the Kit".

Subsequent to our core definition are the results proved in chapters four and five. It is with these that we hope to demonstrate the worth of

our definition. Broadly, our results play one of two important rôles. The simpler results (such as the invariants at the beginning of chapter four) serve to validate the definition of the semantics. If these results didn't hold, there would clearly be something seriously wrong with the definition. Later results, which we would not necessarily *expect* to be true, serve a different rôle. These theorems are either theoretically interesting, or useful in performing verification of programs written in C, or both.

In chapter four we separately prove that two overlapping but distinct sets of expressions are confluent. These results are both theoretically interesting and, as demonstrated by the experiences described in chapter six, pragmatically important as well. An example of a result that is more theoretical in nature is the type preservation property (again from chapter four). However, far from being of theoretical interest only, this result is also a necessary precondition for the more pragmatic confluence results, and many other subsequent results as well.

In chapter five, we prove a suite of results about the semantics of C's statements. In section 5.1 we derive axiomatic style rules for a C programming logic. These results are interesting in part because the various side-conditions required to make the rules hold serve to characterise the degree to which C can be made to look like a cleaner programming language with simple rules. The main omission here is a rule for assignment. Such a rule is difficult to imagine because of the complicated model of memory on top of which it would have to be constructed. In the presence of reliable symbolic evaluation for "straight-line" code, the majority of the programming logic rules we proved are not of much pragmatic interest. However, the rule for loops is useful because it embodies an induction that program verifiers would otherwise have to perform every time they came to verify a loop. In the same chapter, we also prove the automatic loop analysis result of section 5.2, an example of a result where the motivation for the proof was purely pragmatic.

In chapter six, we describe three verification case studies. Two of the case studies were successful verifications of simple programs. The third case study was not successful, being an over-ambitious attempt to verify a larger program. However, this example was attempted first and it was only in attempting it that a number of inadequacies in our model and accompanying tools were revealed. On rectifying these flaws, we were able to move forward and prove the two other results. The verification case studies filled the important rôle of validating the utility of the majority of our verification tools.

Finally, we note that all of this work was only possible with the support of mechanical theorem-proving. While tools such as HOL may be seen (jus-

tifiably or not) to be more of a hindrance than a help in some application domains, the level of detail required in arguments about the semantics of C is such that doing the proofs presented here without HOL's help would have been impossible. Furthermore, even if we supposed that one was capable of doing all of these proofs by hand, others' confidence in the proofs would necessarily suffer in the face of such complexity. Not only does mechanical support reassure one as to the validity of one's proofs, it is also an assurance for others. Thus we are confident both that our proofs are correct, and that others will agree that our theorems are consequences of our definition.

7.1 Future work

There are a number of ways in which this work might be extended and developed upon. Here we aim to elaborate on just a few of them.

Most pressing, we believe, is the need to recast the statement meaning relation ($\rightarrow_s$) as a small-step semantics. For all the superficial convenience of the big-step semantics currently used, its continued use has revealed more and more problems with it. Even with the introduction of a rule for loops with an infinitary hypothesis to model non-termination (something not done in the Cholera mechanisation, though we present the rule in section 3.4.5), the statement semantics will block incorrectly in the presence of a function such as

```
int f(void) { return f(); }
```

Using a small-step semantics for statement evaluation would enable us to do without the special $\mathcal{U}$ marker value, and instead characterise well defined programs simply as those that are guaranteed to evaluate to a value on all possible paths.

This approach would also allow for the simplification of the assignment rule. Currently it keeps track of references made on the RHS's of assignment expressions so that they can be forgotten when the side effect corresponding to the assignment is generated. This needs to happen because reading an object followed by writing to it should be considered undefined. However, we could simplify matters by making only a write followed by a read lead to undefinedness. Then, an expression such as `i = i + 1` can never lead to undefinedness, because the read of `i` on the RHS can never occur after the write to `i`. However, in an expression such as `i + i++`, the path which evaluates the post-increment of `i` would block and we would detect the undefinedness because at least one path would exhibit

the undefined behaviour. This idea is one used by Clive Feather of the C standardisation committee as the basis of a formal semantics for sequence points [Fea98], and may well feature in an Annex to the next revision of the C standard.[1] One problem with this simplification to the treatment of assignment is that while it might simplify characterisation results such as theorem 9 in section 4.4.1, we feel it would likely complicate the analysis necessary for our diamond results, as undefined expressions will no longer be confluent with respect to $\rightarrow_e^*$. (Some paths will lead to blocking or undefinedness, and others will not.)

At the level of the mechanisation, we should also like to bring the treatment of values properly into line with the way they are specified here. Instead of knowing just that we have a family of functions V_τ from byte sequences to values, the Cholera mechanisation assumes that there is an isomorphism between the set of values byte^n, and numbers less than $2^{\texttt{CHAR_BIT}*n}$. This conveniently does away with an irritating source of non-determinism, but is a convenience we may be able to do without. (If we have non-deterministic value representation, then an expression such as `i = 3` may be detectably non-deterministic because of the different choices possible in storing 3 as a sequence of bytes. Determinism results, such as those that appear in chapter four, would need to be re-expressed with a suitable precondition, perhaps simply that the value representation functions for all of the types used in the expression were deterministic.)

Whether or not the model was improved in the ways described above, there are also a few particularly appealing results that we would like to add to those already proved in chapters four and five. The extension of the sequence point free determinism result to what we have termed "sequential expressions" is a particularly obvious example of this. We should also like to see an extension of the notion of syntactic purity so that it included those functions which only included pure expressions and calls to pure functions.

In chapter five, we developed a programming logic for C that implicitly gave a syntactic characterisation of a "safe subset" of the language. This notion does not seem powerful enough on its own, but an appealing extension of this idea would be an analysis that took C programs and proved them in some way equivalent to similar programs running on an abstract machine with a simpler model of state. Such an analysis would need to confirm that variable accesses were all to disjoint objects so that memory could be treated as a finite map from variable names to object values. (The value for an array would be another finite map.) Another form of model

[1] Though it will only be an informative (rather than normative) annex, the same theory explicitly decides whether or not functions may interleave—in the negative.

simplification would be an analysis that confirmed that all operations were on values within their defined domains, thus removing the need for subsequent symbolic evaluation in the simpler domain to assess this possibility.

As far as verification is concerned, we should also like to see the Cholera mechanisation applied to more examples. One appealing possibility would be to follow the example of Yuan Yu's work [Yu93], and verify an implementation of a commonly used library. Needless to say, we should also like to see the BDD example that we attempted completed. Additional verification work would also require the development of better tools. In particular, the symbolic evaluator for expressions that currently relies on rewriting with the definition of the deterministic meaning equivalent should be replaced with a custom-built forward inference system for added efficiency.

So far, these suggestions have been for improvements within the general remit of the thesis. Other possible directions for future work would aim to push beyond those boundaries, extending our model to cope with more kinds of C program. One obvious extension would be to deal with `switch` and `goto` statements. Both of these would be modelled in similar ways, using some sort of label-searching step to find the correct place to resume from within a block of code. (With the `switch` statement, the block of code to search is the body of the statement. With `goto` the block of code to search is the body of the function in which it appears.) Our model would also gain if it were expanded to include the heap, and the possibility of memory allocation that was not lexically determined.

More interesting still however, would be work on extending the model to cope with the concept of an "outside world" and in particular to deal with I/O. A program then becomes something that interacts with its environment as well as something that transforms its internal state. This would naturally suggest trying to characterise C programs with labelled transition systems and verifying infinite behaviours with tools such as bisimilarity. We would thus extend the scope of our work to be able to accurately describe C programs that run in a concurrent environment, ranging from client-server interactions, to implementations of security protocols, to a wide range of Internet tools and applications.

Finally, it is worth noting that the programming language C++ has a great deal in common with C. The newer language, C++ split from C in the early 1980s and evolved very rapidly, adding a number of new features. C evolved more gradually and is, with a few minor exceptions, a proper subset of C++. This suggests that another extension of our work would be to model the extra complexities of C++. Two features that would be particularly interesting are C++'s object orientation, and its form of parametric polymorphism, templates. C++ is about to have its first standard pub-

lished as we write and would surely be a fascinating challenge to describe and analyse formally.

Concluding remark

C is a powerful, ubiquitous programming language that is an important enabling technology for much of computer science's impact on today's world. With the help of mechanical theorem-proving assistance, we have demonstrated that the theory of formal methods and programming language semantics *can* be fruitfully applied to real world examples such as C. We hope that our example will inspire others to attack similar real-world problems with emerging theorem-proving technology.

Appendix A

Definitions

A.1 Syntax

The type of C types is defined thus:

```
val basic_integral = define_type {
  name = "basic_integral_type",
  fixities = [Prefix, Prefix, Prefix, Prefix],
  type_spec =
    `basic_integral_type = Char | Short | Int | Long`};
local
  val type_name = "CType"
  val bi_type = (==`:basic_integral_type`==)
  structure CTypeDef : NestedRecTypeInputSig =
  struct
    structure DefTypeInfo = DefTypeInfo
    open DefTypeInfo

    val def_type_spec =
       [{type_name = type_name,
         constructors =
           [{name = "Void", arg_info = []},
            {name = "Unsigned", arg_info = [existing bi_type]},
            {name = "Signed", arg_info = [existing bi_type]},
            {name = "Ptr",  arg_info = [being_defined "CType"]},
            {name = "Array", arg_info = [being_defined "CType",
                                         existing (==`:num`==)]},
            {name = "Struct",
             arg_info = [existing (==`:string`==)]},
            {name = "Function",
             arg_info = [being_defined "CType", (* the return type *)
                         type_op {Tyop = "list",(* parameters      *)
                                  Args = [being_defined "CType"]}]}]}]
```

Type well-formedness is defined as follows (where no_dups checks that an list of name-type pairs doesn't duplicate a name):

```
local
  val wf = ``wf_type:(string -> str_info) ->
                             (string -> bool) ->
                             CType -> bool``
  fun s t = {hypotheses = [], side_conditions = [], conclusion = t}
in
  val (wf_type_rules,wf_type_induction,wf_type_cases) =
  IndDefLib.new_inductive_definition {
    fixity = Prefix,
    name = "wf_type",
    patt = (``^wf smap vstd t``, []:term list),
    rules = [
      s ``^wf smap vstd Void``,
      s ``^wf smap vstd (Unsigned bt)``,
      s ``^wf smap vstd (Signed bt)``,
      {hypotheses = [``^wf smap vstd pt``],
       side_conditions = [],
       (* ------------------------------------------ *)
       conclusion = ``^wf smap vstd (Ptr pt)``},

      {hypotheses = [],
       side_conditions = [``(sn:string) IN vstd``],
       conclusion = ``^wf smap vstd (Ptr (Struct sn))``},

      {hypotheses = [``^wf smap vstd bt``],
       side_conditions = [``~(n = 0)``, ``~(bt = Void)``],
       (* ------------------------------------------ *)
       conclusion = ``^wf smap vstd (Array bt n)``},
      {hypotheses = [
         ``!t. IS_EL t (MAP SND (struct_info (smap (sn:string)))) ==>
               ^wf smap (sn INSERT vstd) t``
       ],
       side_conditions = [
         ``nodup_flds (smap (sn:string)) /\ ~(struct_info (smap sn) = []) /\
           !t. IS_EL t (MAP SND (struct_info (smap sn))) ==> ~(t = Void)``
       ],
       (* ---------------------------------------------------- *)
       conclusion = ``^wf smap vstd (Struct sn)``},
      {hypotheses = [
         ``^wf smap vstd rtype``,
         ``!t. IS_EL t args ==>
               ^wf smap vstd t /\ ~(t = Void) /\ ~array_type t``
       ],
       side_conditions = [``~array_type rtype``],
       (*---------------------------------------------- *)
       conclusion =
         ``^wf smap vstd (Function rtype args)``}
```

```
    ]}
end;
```

The type of expressions:

```
local
  val existing   = DefTypeInfo.existing
  val being_defined = DefTypeInfo.being_defined
  val numtype    = existing (==`:num`==)
  val oparg      = ty_antiq (==`:^memval # CType # num`==)
  val resarg     = ty_antiq (==`:^memval # CType`==)
  val stringtype = existing (==`:string`==)   and
      ctypetype  = existing (==`:CType`==)    and
      cbinfntype = existing ``:c_binops``     and
      cunfntype  = existing ``:c_unops``      and
      memvaltype = existing (==`:^memval`==)  and
      membagtype = existing (==`:num->num`==) and
      cexprtype  = being_defined "CExpr"      and
      type_name  = "CExpr"
  structure CExprDef : NestedRecTypeInputSig =
  struct
    structure DefTypeInfo = DefTypeInfo
    open DefTypeInfo
    val def_type_spec =
      [{type_name = type_name,
        constructors =
          [{name = "Cnum",      arg_info = [numtype]},
           {name = "Cchar",     arg_info = [numtype]},
           {name = "Cnullptr",  arg_info = [ctypetype]},
           {name = "Var",       arg_info = [stringtype]},
           {name = "CFunRef",   arg_info = [stringtype]},
           {name = "COr",       arg_info = [cexprtype, cexprtype]},
           {name = "CAnd",      arg_info = [cexprtype, cexprtype]},
           {name = "CCond",     arg_info = [cexprtype, cexprtype,
                                            cexprtype]},
           {name = "CApBinary", arg_info = [cbinfntype, cexprtype,
                                            cexprtype]},
           {name = "CApUnary",  arg_info = [cunfntype, cexprtype]},
           {name = "Deref",     arg_info = [cexprtype]},
           {name = "Addr",      arg_info = [cexprtype]},
           {name = "Assign",    arg_info = [cbinfntype, cexprtype,
                                            cexprtype, membagtype]},
           {name = "SVar",      arg_info = [cexprtype, stringtype]},
           {name = "FnApp",     arg_info = [cexprtype,
                      type_op {Tyop = "list", Args = [cexprtype]}]},
           {name = "CommaSep",  arg_info = [cexprtype, cexprtype]},
           {name = "Cast",      arg_info = [ctypetype, cexprtype]},
           {name = "PostInc",   arg_info = [cexprtype]},
           {name = "CAndOr_sqpt",  arg_info = [cexprtype]},
           {name = "FnApp_sqpt", arg_info = [cexprtype,
                                             type_op {Tyop = "list",
                                                      Args = [cexprtype]}]},
```

```
{name = "LVal", arg_info = [numtype, ctypetype]},
{name = "RValreq", arg_info = [cexprtype]},
{name = "ECompVal",  arg_info = [memvaltype, ctypetype]},
{name = "UndefinedExpr", arg_info = []}]}]
```

A.2 Semantics

The states over which the semantics operates are defined in two parts. First, the core record type (with the global versions of components indicated by the leading "g" in the name):

```
create_record "fn_info"[("return_type", (==`:CType`==)),
                        ("parameters",  (==`:(string#CType) list`==)),
                        ("body",        (==`:CStmt`==))];
create_record "CState" [("allocmap",  (==`:num -> bool`==)),
                        ("fnmap",     (==`:string -> fn_info`==)),
                        ("gstrmap",   (==`:string -> str_info`==)),
                        ("gtypemap",  (==`:string -> CType`==)),
                        ("gvarmap",   (==`:string -> num`==)),
                        ("initmap",   (==`:num -> bool`==)),
                        ("locmap",    (==`:num -> MemObj`==)),
                        ("strmap",    (==`:string -> str_info`==)),
                        ("typemap",   (==`:string -> CType`==)),
                        ("varmap",    (==`:string -> num`==))];
```

Then we define a side effect record type. This contains the three state components that are particular to expression side effects. It is a historical accident that Cholera defines this part of the state separately: the false optimisation of realising that side effect records never had a rôle to play in statement evaluation lead us to create a separate "bit of state" that was only in existence when expressions were evaluated. Sad to say, mechanisations can be difficult to reorganise once established in numerous files and theories.

```
create_record "se_info" [("pending_ses", ==`:^qse->num`==),
                         ("update_map", ==`:num->bool`==),
                         ("ref_map", ==`:num->num`==)];
```

We define our $\mathcal{E}$ and $\mathcal{L}$ contexts as functions taking expressions to expressions.

```
val valid_econtext = new_definition(
  "valid_econtext",
  --`valid_econtext f =
      (?f' e1. f = CApBinary f' e1) \/
      (?f' e2. f = \e1. CApBinary f' e1 e2) \/
      (?f' e2 b. f = \e1. Assign f' e1 e2 b) \/
      (?e2 f'. (f = (\e1. f' e1 e2)) /\
               f' IN {COr; CAnd; CommaSep}) \/
      (?e2 e3. (f = \e1. CCond e1 e2 e3)) \/
      (?f'. f = CApUnary f') \/
      (f IN  {Addr; Deref; CAndOr_sqpt; PostInc; RValreq}) \/
      (?fld. f = \e. SVar e fld) \/
      (?args. f = \e. FnApp e args) \/
      (?args fn n. (f = \e. FnApp fn (LIST_INSERT e n args)) /\
                   (n <= LENGTH args)) \/
      (?t. f = Cast t)`--);
```

```
val valid_lvcontext = new_definition (
  "valid_lvcontext",
  --`valid_lvcontext f =
        (?f' e1. f = CApBinary f' e1) \/
        (?f' e2. f = \e1. CApBinary f' e1 e2) \/
        (?e2 f'. (f = (\e1. f' e1 e2)) /\
                 f' IN {COr; CAnd; CommaSep}) \/
        (?e2 e3. (f = \e1. CCond e1 e2 e3)) \/
        (?f'. f = CApUnary f') \/
        (f IN  {Deref; CAndOr_sqpt; RValreq}) \/
        (?args. f = \e. FnApp e args) \/
        (?args fn n. (f = \e. FnApp fn (LIST_INSERT e n args)) /\
                     (n <= LENGTH args)) \/
        (?t. f = Cast t)`--);
```

The following are helper definitions for the definition of `meaning` the dynamic reduction relation. The most important is the first, `lval2rval`, which takes an lvalue to a real value. The `malloc` relation defined next is referred to as `alloc` in the text.

```
val lval2rval = new_definition(
  "lval2rval",
  ``lval2rval (s0,e0,se0) (s,e,se) =
       (s0 = s) /\
       ?n t. (e0 = LVal n t) /\
             (~(array_type t) /\
              (?sz. sizeof (strmap s0) (INL t) sz /\
                    (mark_ref n sz se0 se /\
                     (range_set n sz) SUBSET initmap s0 /\
                     (e = ECompVal (mem2val s0 n sz) t) \/
                     (~(range_set n sz SUBSET initmap s0) \/
                      (!se'. ~(mark_ref n sz se0 se'))) /\
                     (se = se0) /\ (e = UndefinedExpr))) \/
              (?sz t'.
                  (t = Array t' sz) /\ (se0 = se) /\
                  (e = ECompVal (num2mval ptr_size n) (Ptr t'))))``);
                theorem "choltype" "CType_one_one"]);

val malloc = new_definition(
  "malloc",
  ``malloc s0 a n =
       DISJOINT (allocmap s0) (range_set a n) /\
       ~(a = 0) /\
       a + n < 2 EXP (CHAR_BIT * ptr_size)``);
val rec_i_vars = Rsyntax.new_recursive_definition {
  name = "rec_i_vars", fixity = Prefix,
  rec_axiom = theorem "list" "list_Axiom",
  def =
   ``(rec_i_vars st1 [] st2 resv = (st1 = st2) /\ (resv = T)) /\
     (rec_i_vars st1 (CONS (hd:string#CType) tail) st2 resv =
```

```
        ?n.
           sizeof (strmap st1) (INL (SND hd)) n /\
           ((?a. malloc st1 a n /\
                 rec_i_vars
                   ((varmap_fupd  (\v. override v (FST hd) a) o
                     typemap_fupd (\t. override t (FST hd) (SND hd)) o
                     allocmap_fupd ($UNION (range_set a n))) st1)
                   tail
                   st2 resv) \/
           (!a. ~malloc st1 a n) /\ (resv = F) /\ (st2 = st1)))``
};
val install_vars = new_definition(
  "install_vars",
  ``install_vars st1 fn st2 resv =
        rec_i_vars ((varmap_update (gvarmap st1) o
                     typemap_update (gtypemap st1) o
                     strmap_update (gstrmap st1)) st1)
                   (parameters (fnmap st1 fn))
                   st2 resv``);
val rec_i_vals = Rsyntax.new_recursive_definition {
  name = "rec_i_vals", fixity = Prefix,
  rec_axiom = theorem "list" "list_Axiom",
  def = (--`
    (rec_i_vals st1 [] vallist st2 res =
       (vallist = []) /\ (st1 = st2) /\ res) /\
    (rec_i_vals s0 (CONS (phd:string#CType) ptl) vallist s res =
       ?vval vtype vtl pname ptype.
          (vallist = CONS (ECompVal vval vtype) vtl) /\
          (phd = (pname, ptype)) /\
          ((?s1 newval rs.
             convert_val (strmap s0) (vval, vtype) (newval, ptype) /\
             (rs = range_set (varmap s0 pname) (LENGTH newval)) /\
             (s1 = initmap_fupd ($UNION rs)
                     (allocmap_fupd ($UNION rs)
                        (val2mem s0 (varmap s0 pname) newval))) /\
             rec_i_vals s1 ptl vtl s res) \/
           (!nv. ~convert_val (strmap s0) (vval,vtype) (nv, ptype)) /\
           (res = F)))`--)};
val install_values = new_definition (
  "install_values",
  ``install_values s0 fn pvl s1 res =
          rec_i_vals s0 (parameters (fnmap s0 fn)) pvl s1 res``);
val pass_parameters = new_definition (
  "pass_parameters",
  ``pass_parameters s0 fnid pv s res =
      ?s1 res'.
        install_vars s0 fnid s1 res' /\
        (res'
         => install_values s1 fnid pv s res
```

```
        |  (res = F) /\ (s = s0))``);
```

Finally we define the meaning relation itself. This has to be cast as six mutually recursive relations. The meaningfuls type is the disjoint union type which determines which relation is intended for the various syntactic forms. For example, mExpr tags for $\rightarrow_e$, mTCExpr for $\rightarrow_e^*$, and mStmt for $\rightarrow_s$. The meaning_val type is the disjoint sum of possible result types. The ExprVal constructor is used to tag all results of the $\rightarrow_e$ and $\rightarrow_e^*$ relations. The other constructors are all the different possible results for statement and variable declaration evaluation.

```
val meaningfuls = define_type {
  name = "meaningfuls",
  fixities = [Prefix,Prefix,Prefix,Prefix,Prefix,Prefix],
  type_spec = `meaningfuls =
                mExpr of CExpr => se_info |
                mTCExpr of CExpr => se_info |
                mStmt of CStmt | mStmtl of CStmt list |
                mVarD of var_decl | mVarDl of var_decl list`
};

val meaning_val = define_type {
  name = "meaning_val_Axiom",
  fixities =
    [Prefix, Prefix, Prefix, Prefix, Prefix, Prefix, Prefix, Prefix],
  type_spec = `meaning_val = ExprVal of CExpr => se_info |
                             StmtVal | RetVal of MemObj list |
                             BreakVal | ContVal | VarDeclVal |
                             Undefined`};

local
  val mng  =
    (--`meaning:meaningfuls -> CState ->
                (CState # meaning_val) -> bool`--)
  val ev = (--`ExprVal`--)
  val evl = (--`FunArgsVal`--)
  fun s x = (* s for simple *)
     {conclusion = x, side_conditions = [], hypotheses = []}
  val ind_definition = {
    fixity = Prefix,
    name = "meaning",
    patt = ((--`^mng x s sv`--), ([]:term list)),
    rules = [
      s (--`^mng (mExpr (Cnum n) se) s
                (s, ^ev (ECompVal (num2mval int_size n)
                                  (Signed Int))
                        se)`--),
      s (--`^mng (mExpr (Cchar n) se) s
                (s, ^ev (ECompVal (num2mval int_size n)
                                  (Signed Int))
```

```
                   se)`--),
s (--`^mng (mExpr (Cnullptr t) se) s
           (s, ^ev (ECompVal (num2mval ptr_size 0) (Ptr t)) se)`--),
s (--`^mng (mExpr (CFunRef n) se) s
           (s, ^ev (ECompVal (create_memval_fnref n)
                             (typemap s n))
                   se)`--),
s (--`^mng (mExpr (Var vname) se) s
           (s, ^ev (LVal (varmap s vname) (typemap s vname)) se)`--),
{hypotheses = [],
 side_conditions = [``convert_val (strmap s) (v, t) (v', t')``],
 conclusion = ``^mng (mExpr (Cast t' (ECompVal v t)) se) s
                     (s, ^ev (ECompVal v' t') se)``},
{hypotheses = [],
 side_conditions = [``!v'. ~convert_val (strmap s) (v, t) (v', t')``],
 (* ---------------------------------------------------------- *)
 conclusion = ``^mng (mExpr (Cast t' (ECompVal v t)) se) s
                     (s, ^ev UndefinedExpr se)``},
{hypotheses = [(--`^mng (mExpr e se0) s0 (s, ^ev e' se)`--)],
 side_conditions = [(--`valid_econtext f`--)],
 (* ------------------------------------------------------- *)
 conclusion = (--`^mng (mExpr ((f:CExpr->CExpr) e) se0) s0
                       (s, ^ev (f e') se)`--)},
{hypotheses = [],
 side_conditions = [
   ``valid_econtext f \/ ?asfn lhs b. f = \e. Assign asfn lhs e b``
 ],
 conclusion = (--`^mng (mExpr (f UndefinedExpr) se) s
                       (s, ^ev UndefinedExpr se)`--)},
{hypotheses = [],
 side_conditions = [--`valid_lvcontext f`--,
                    --`lval2rval (s0,e0,se0) (s,e,se)`--],
 conclusion = --`^mng (mExpr ((f:CExpr->CExpr) e0) se0) s0
                      (s, ^ev (f e) se)`--},
{hypotheses = [],
 side_conditions = [(--`apply_se (se0, s0) (se, s)`--)],
 conclusion = (--`^mng (mExpr e se0) s0 (s, ^ev e se)`--)},
{hypotheses = [],
 side_conditions = [(--`!se s. ~(apply_se (se0, s0) (se, s))`--),
                    ``~is_null_se se0``, ``~(e = UndefinedExpr)``],
 conclusion =
   (--`^mng (mExpr e se0) s0 (s0, ^ev UndefinedExpr se0)`--)},
{hypotheses = [],
 side_conditions = [(--`is_null_se se`--)],
 (* ------------------------------------------------------- *)
 conclusion = (--`^mng (mExpr (CommaSep (ECompVal v t) e2) se) s0
                       (s0, ^ev (RValreq e2) (null_se se))`--)},
s (--`^mng (mExpr (RValreq (ECompVal v t)) se) s
           (s, ^ev (ECompVal v t) se)`--),
```

```
{hypotheses = [],
 side_conditions = [
   ``!res restype.
        ~(binop_meaning s f v1 type1 v2 type2 res restype)``
 ],
 (* ------------------------------------------------------------ *)
 conclusion =
   (--`^mng (mExpr (CApBinary f (ECompVal v1 type1)
                               (ECompVal v2 type2)) se0) s
            (s, ^ev UndefinedExpr se0)`--)},
{hypotheses = [],
 side_conditions = [
   ``binop_meaning s f v1 type1 v2 type2 res restype``
 ],
 (* ----------------------------------------------------------- *)
 conclusion =
   (--`^mng (mExpr (CApBinary f (ECompVal v1 type1)
                               (ECompVal v2 type2)) se) s
            (s, ^ev (ECompVal res restype) se)`--)},
{hypotheses = [],
 side_conditions = [
   --`unop_meaning s f ival t result rt`--
 ],
 (* --------------------------------------------------------- *)
 conclusion = (--`^mng (mExpr (CApUnary f (ECompVal ival t)) se) s
                       (s, ^ev (ECompVal result rt) se)`--)},
{hypotheses = [],
 side_conditions = [
   --`!res rt. ~(unop_meaning s0 f ival t res rt)`--
 ],
 (* ------------------------------------------------------------- *)
 conclusion = (--`^mng (mExpr (CApUnary f (ECompVal ival t)) se0) s0
                       (s0, ^ev UndefinedExpr se0)`--)},
{hypotheses = [],
 side_conditions = [(--`coerce_to_num v = 0`--),
                    (--`scalar_type t`--)],
 (* ------------------------------------------------------------- *)
 conclusion = --`
   ^mng (mExpr (CAnd (ECompVal v t) sub2) se) s
        (s, ^ev (ECompVal (num2mval int_size 0) (Signed Int)) se)`--},
{hypotheses = [],
 side_conditions = [(--`~(coerce_to_num v = 0)`--),
                    (--`is_null_se se`--), (--`scalar_type t`--)],
 (* ------------------------------------------------------------- *)
 conclusion = (--`^mng (mExpr (CAnd (ECompVal v t) sub2) se) s
                       (s, ^ev (CAndOr_sqpt sub2) (null_se se))`--)},
{hypotheses = [],
 side_conditions = [(--`scalar_type t`--)],
 (* ------------------------------------------------------------- *)
```

```
 conclusion =
   (--`^mng (mExpr (CAndOr_sqpt (ECompVal v t)) se) s
            (s, ^ev (ECompVal (coerce_to_num v = 0
                               => (num2mval int_size 0)
                               |  (num2mval int_size 1))
                              (Signed Int)) se)`--)},
{hypotheses = [],
 side_conditions = [(--`~(coerce_to_num v = 0)`--),
                    (--`scalar_type t`--)],
 (* ---------------------------------------------------- *)
 conclusion = --`
   ^mng (mExpr (COr (ECompVal v t) sub2) se) s
        (s, ^ev (ECompVal (num2mval int_size 1) (Signed Int)) se)`--},
{hypotheses = [],
 side_conditions = [(--`(coerce_to_num v = 0)`--),
                    (--`is_null_se se`--), (--`scalar_type t`--)],
 (* ----------------------------------------------------------- *)
 conclusion = (--`^mng (mExpr (COr (ECompVal v t) sub2) se) s
                       (s, ^ev (CAndOr_sqpt sub2) (null_se se))`--)},
{hypotheses = [],
 side_conditions = [
   ``is_null_se se``, ``scalar_type t``,
   ``expr_type (expr_type_comps s) RValue (INL e2) (INL t2)``,
   ``expr_type (expr_type_comps s) RValue (INL e3) (INL t3)``,
   ``expr_type (expr_type_comps s) RValue
               (INL (CCond (ECompVal v t) e2 e3))
               (INL result_type)``,
   ``coerce_to_num v = 0``,
   ``(t2 = Struct sn) /\ (resexpr = RValreq e3) \/
     (!sn. ~(t2 = Struct sn)) /\ (resexpr = Cast result_type e3)``
 ],
 (* ---------------------------------------------------------- *)
 conclusion = --`^mng (mExpr (CCond (ECompVal v t) e2 e3) se) s
                      (s, ^ev resexpr (null_se se))`--},

{hypotheses = [],
 side_conditions = [
   ``is_null_se se``, ``scalar_type t``,
   ``expr_type (expr_type_comps s) RValue (INL e2) (INL t2)``,
   ``expr_type (expr_type_comps s) RValue (INL e3) (INL t3)``,
   ``expr_type (expr_type_comps s) RValue
               (INL (CCond (ECompVal v t) e2 e3))
               (INL result_type)``,
   ``~(coerce_to_num v = 0)``,
   ``(t2 = Struct sn) /\ (resexpr = RValreq e2) \/
     (!sn. ~(t2 = Struct sn)) /\ (resexpr = Cast result_type e2)``
 ],
 (* ----------------------------------------------------------- *)
 conclusion = --`^mng (mExpr (CCond (ECompVal v t) e2 e3) se) s
```

```
                    (s, ^ev resexpr (null_se se))`--},
{hypotheses = [],
 side_conditions = [--`~(t = Void)`--],
 conclusion = --`^mng (mExpr (Deref (ECompVal mval (Ptr t))) se) s
                    (s, ^ev (LVal (memval2addr mval) t) se)`--},
s (--`^mng (mExpr (Addr (LVal a t)) se) s
          (s, ^ev (ECompVal (num2mval ptr_size a) (Ptr t)) se)`--),
{hypotheses = [(--`^mng (mExpr RHS se0) s0 (s, ^ev e se)`--)],
 side_conditions = [
   (--`mb' = BAG_delta (ref_map se0, ref_map se) mb`--)],
 (* ------------------------------------------------------------ *)
 conclusion =
   (--`^mng (mExpr (Assign f a RHS mb) se0) s0
           (s, ^ev (Assign f a e mb') se)`--)},
{hypotheses = [],
 side_conditions = [``~(f = CAssign)``],
 conclusion =
   ``^mng (mExpr (Assign f (LVal n t) e mb) se0) s0
         (s0, ExprVal (Assign CAssign
                       (LVal n t)
                       (CApBinary f (LVal n t) e)
                       mb)
                     se0)``},
{hypotheses = [],
 side_conditions = [
     ``convert_val (strmap s) (v0,t0) (v,lhs_t) /\
       (ok_refs = \x. x IN (se_affects (a, v)) => mb x | 0) /\
       (se' = ref_map_fupd (\rm. BAG_DIFF rm ok_refs) se0) /\
       (se = add_se (a, v) se') /\ (resv = ECompVal v lhs_t)
                         \/
       (!v. ~convert_val (strmap s) (v0, t0) (v, lhs_t)) /\
       (resv = UndefinedExpr) /\ (se = se0)``
 ],
 (* ------------------------------------------------------------ *)
 conclusion = ``^mng (mExpr (Assign CAssign (LVal a lhs_t)
                                           (ECompVal v0 t0)
                                           mb)
                          se0) s (s, ^ev resv se)``},
{hypotheses = [],
 side_conditions = [
   --`sizeof (strmap s) (INL t) sz`--,
   --`v = mem2val s a sz`--,
   ``range_set a sz SUBSET (initmap s)``,
   --`binop_meaning s CPlus v t
                          (num2mval int_size 1) (Signed Int)
                          nv1 t'`--,
   ``convert_val (strmap s) (nv1,t') (nv,t) /\
     (se = add_se (a, nv) se0) /\ (resv = ECompVal v t)
             \/
```

```
    (!nv. ~convert_val (strmap s) (nv1, t') (nv, t)) /\
    (se = se0) /\ (resv = UndefinedExpr)``
 ],
 (* ------------------------------------------------------------- *)
 conclusion =
   (--`^mng (mExpr (PostInc (LVal a t)) se0) s (s, ^ev resv se)`--)},
{hypotheses = [],
 side_conditions = [
   ``sizeof (strmap s) (INL t) sz``,
   ``v = mem2val s a sz``,
   ``(!nv1 t'.
        ~binop_meaning s CPlus v t (num2mval int_size 1) (Signed Int)
                                   nv1 t') \/
    ~(range_set a sz SUBSET (initmap s))``
 ],
 (* ---------------------------------------------------------------- *)
 conclusion = ``^mng (mExpr (PostInc (LVal a t)) se0) s
                    (s, ^ev UndefinedExpr se0)``},
{hypotheses = [],
 side_conditions = [
   --`offset (strmap s) st fld offn`--,
   --`lookup_field_info (strmap s st) fld ftype`--
 ],
 (* ---------------------------------------------------- *)
 conclusion =
   --`^mng (mExpr (SVar (LVal a (Struct st)) fld) se) s
          (s, ^ev (LVal (a + offn) ftype) se)`--},

{hypotheses = [],
 side_conditions = [
   --`offset (strmap s) st fld offn`--,
   --`lookup_field_info (strmap s st) fld ftype`--,
   --`sizeof (strmap s) (INL ftype) fsz`--,
   --`(fv:^memval) = GENLIST (\n. EL (n + offn) v) fsz`--
 ],
 (* ---------------------------------------------------------------- *)
 conclusion =
   (--`^mng (mExpr (SVar (ECompVal v (Struct st)) fld) se) s
           (s, ^ev (ECompVal fv ftype) se)`--)},
{hypotheses = [],
 side_conditions = [
   --`ALL_EL (\e. ?v t. e = ECompVal v t) (CONS f params)`--,
   --`is_null_se se`--],
 conclusion =
   --`^mng (mExpr (FnApp f params) se) s
          (s, ^ev (FnApp_sqpt f params) (null_se se))`--},
{hypotheses = [
   (--`^mng (mStmt (body (fnmap s1 fnid))) s1 (s2, rv)`--)],
 side_conditions = [
```

```
    (--`pass_parameters s0 fnid params s1 T`--),
    (--`memval_fnref fnval fnid`--),
    (--`ftype = Function rt vs`--),
    (--`(?v. (rv = RetVal v) /\ (ev = ECompVal v rt) /\
             (s = locmap_update (locmap s2)
                    (initmap_update (initmap s2 INTER allocmap s0)
                                    s0))) \/
        (rv = Undefined) /\ (ev = UndefinedExpr) /\ (s = s0)`--)
  ],
  (* ------------------------------------------------------------- *)
  conclusion =
     (--`^mng (mExpr (FnApp_sqpt (ECompVal fnval ftype) params) se) s0
              (s, ^ev ev se)`--)},
 {hypotheses = [],
  side_conditions = [
    ``pass_parameters s0 fnid params s F``,
    ``memval_fnref fnval fnid``
  ],
  conclusion =
    ``^mng (mExpr (FnApp_sqpt (ECompVal fnval ftype) params) se) s0
           (s0, ^ev UndefinedExpr se)``},
 s (--`^mng (mTCExpr e se) s (s, ^ev e se)`--),

 {hypotheses = [
   (--`^mng (mExpr e0 se0) s0 (s', ^ev e' se')`--),
   (--`^mng (mTCExpr e' se') s' (s, ^ev e se)`--)
  ],
  side_conditions = [],
  (* ---------------------------------------------------------- *)
  conclusion = (--`^mng (mTCExpr e0 se0) s0 (s, ^ev e se)`--)},
 s (--`^mng (mStmt EmptyStmt) s (s, StmtVal)`--),

 s (--`^mng (mStmtl []) s0 (s0, StmtVal)`--),

 {hypotheses = [
    (--`^mng (mStmt st1) s1 (s', StmtVal)`--),
    (--`^mng (mStmtl sttail) s' (s2, v)`--)
  ],
  side_conditions = [],
  (* -------------------------------------------------------- *)
  conclusion = (--`^mng (mStmtl (CONS st1 sttail)) s1 (s2,v)`--)},

 {hypotheses = [(--`^mng (mStmt st1) s1 (s', v)`--)],
  side_conditions = [(--`~(v = StmtVal)`--)],
  (* --------------------------------------------------------- *)
  conclusion =
    (--`^mng (mStmtl (CONS st1 sttail)) s1 (s', v)`--)},

 {hypotheses = [
```

```
    (--`^mng (mTCExpr (RValreq e) base_se) s1 (s2, ^ev e' se)`--)
  ],
  side_conditions = [
    (--`is_null_se se`--),
    (--`(e' = ECompVal v t) /\ (retval = RetVal v) \/
        (e' = UndefinedExpr) /\ (retval = Undefined)`--)],
  (* ------------------------------------------------- *)
  conclusion = (--`^mng (mStmt (Ret e)) s1 (s2, retval)`--)},

 s (--`^mng (mStmt EmptyRet) s1 (s1, RetVal [])`--),
 s (--`^mng (mStmt Break) s1 (s1, BreakVal)`--),
 s (--`^mng (mStmt Cont) s1 (s1, ContVal)`--),
 {hypotheses = [--`^mng (mStmt st) s0 (s, v)`--],
  side_conditions = [--`traplink tt v`--],
  conclusion = --`^mng (mStmt (Trap tt st)) s0 (s, StmtVal)`--},

 {hypotheses = [--`^mng (mStmt st) s0 (s, v)`--],
  side_conditions = [--`~(traplink tt v)`--],
  conclusion = --`^mng (mStmt (Trap tt st)) s0 (s, v)`--},

 {hypotheses = [
    --`^mng (mTCExpr (RValreq exp) base_se) s1 (s2, ^ev val se)`--],
  side_conditions = [
    --`(val = ECompVal v t) /\ (retval = StmtVal) /\ is_null_se se \/
       (val = UndefinedExpr) /\ (retval = Undefined)`--
  ],
  (* ------------------------------------------------------------ *)
  conclusion = (--`^mng (mStmt (Standalone exp)) s1 (s2, retval)`--)},
 {hypotheses = [
    --`^mng (mTCExpr (RValreq guard) base_se) s0
            (s, ^ev UndefinedExpr se)`--],
  side_conditions = [],
  conclusion = (--`^mng (mStmt (CIf guard t e)) s0 (s, Undefined)`--)},

 {hypotheses = [
    (--`^mng (mTCExpr (RValreq guard) base_se) s1
             (s', ^ev (ECompVal gval t) se)`--),
    (--`^mng (mStmt then) s' (s2, val)`--)
  ],
  side_conditions = [
    (--`~(coerce_to_num gval = 0)`--),  (* guard is true *)
    (--`scalar_type t`--),
    (--`is_null_se se`--)
  ],
  (* ------------------------------------------------------- *)
  conclusion =
    (--`^mng (mStmt (CIf guard then else)) s1 (s2, val)`--)},

 {hypotheses = [
```

```
    (--`^mng (mTCExpr (RValreq guard) base_se) s1
             (s', ^ev (ECompVal gval t) se)`--),
    (--`^mng (mStmt else) s' (s2, val)`--)
  ],
  side_conditions = [
    (--`(coerce_to_num gval = 0)`--), (* guard is false *)
    (--`scalar_type t`--),
    (--`is_null_se se`--)
  ],
  (* -------------------------------------------------- *)
  conclusion =
    (--`^mng (mStmt (CIf guard then else)) s1 (s2, val)`--)},
 {hypotheses = [
    --`^mng (mTCExpr (RValreq guard) base_se) s0
            (s, ^ev UndefinedExpr se)`--],
  side_conditions = [],
  (* ------------------------------------------------------------ *)
  conclusion =
    (--`^mng (mStmt (CLoop guard bdy)) s0 (s, Undefined)`--)},

 {hypotheses = [
    (--`^mng (mTCExpr (RValreq guard) base_se) s0
             (s, ^ev (ECompVal gval t) se)`--)
  ],
  side_conditions = [(--`scalar_type t`--),
                     (--`coerce_to_num gval = 0`--),
                     (--`is_null_se se`--)],
  (* ------------------------------------------------------------ *)
  conclusion =
    (--`^mng (mStmt (CLoop guard bdy)) s0 (s, StmtVal)`--)},
 {hypotheses = [
    (--`^mng (mTCExpr (RValreq guard) base_se) s0
             (s', ^ev (ECompVal gval t) se)`--),
    (--`^mng (mStmt bdy) s' (s, v)`--)
  ],
  side_conditions = [
    (--`~(v = StmtVal)`--), (--`scalar_type t`--),
    (--`~(coerce_to_num gval = 0)`--), (--`is_null_se se`--)
  ],
  (* ---------------------------------------------------------- *)
  conclusion =
    (--`^mng (mStmt (CLoop guard bdy)) s0 (s, v)`--)},
 {hypotheses = [
    (--`^mng (mTCExpr (RValreq guard) base_se) s0
             (s', ^ev (ECompVal gval t) se)`--),
    (--`^mng (mStmt bdy) s' (s'', StmtVal)`--),
    (--`^mng (mStmt (CLoop guard bdy)) s'' (s, v)`--)
  ],
  side_conditions = [
```

```
  (--`scalar_type t`--), (--`~(coerce_to_num gval = 0)`--),
  (--`is_null_se se`--)
 ],
 (* ------------------------------------------------------------ *)
 conclusion = (--`^mng (mStmt (CLoop guard bdy)) s0 (s, v)`--)},
{hypotheses = [
  (--`^mng (mVarDl vds) s0 (s1, VarDeclVal)`--),
  (--`^mng (mStmtl sts) s1 (s2, v)`--)
 ],
 side_conditions = [],
 (* ------------------------------------------ *)
 conclusion =
  (--`^mng (mStmt (Block vds sts))
           s0
           (locmap_update (locmap s2)
              (initmap_update
                 (initmap s2 INTER allocmap s0)
                 s0), v)`--)},
{hypotheses = [(--`^mng (mVarDl vds) s0 (s, Undefined)`--)],
 side_conditions = [],
 (* --------------------------------------------------------------- *)
 conclusion = (--`^mng (mStmt (Block vds sts)) s0 (s0, Undefined)`--)},
{hypotheses = [],
 side_conditions = [
  ``sizeof (strmap s) (INL type) n``,
  ``malloc s a n``
 ],
 (* ------------------------------------------------------------ *)
 conclusion =
  (--`^mng (mVarD (VDec type name))
           s
           ((allocmap_fupd ($UNION (range_set a n)) o
             varmap_fupd (\v. override v name a)    o
             typemap_fupd (\t. override t name type)) s,
            VarDeclVal)`--)},

{hypotheses = [],
 side_conditions = [``!a. ~malloc s a (sizeof_fn (strmap s) type)``],
 (* -------------------------------------------------------------- *)
 conclusion = ``^mng (mVarD (VDec type name)) s (s, Undefined)``},
{hypotheses = [
  ``^mng (mVarD (VDec t name)) s0 (s1, VarDeclVal)``,
  ``^mng (mTCExpr
           (Assign CAssign (Var name) (RValreq e) EMPTY_BAG) base_se)
          s1 (s, ^ev (ECompVal v t) se)``
 ],
 side_conditions = [``is_null_se se``],
 (* ------------------------------------------------- *)
 conclusion =
```

```
        (--`^mng (mVarD (VDecInit t name e)) s0 (s, VarDeclVal)`--)},
      {hypotheses = [
        ``^mng (mVarD (VDec t name)) s0 (s1, VarDeclVal)``,
        ``^mng (mTCExpr
                  (Assign CAssign (Var name)(RValreq e) EMPTY_BAG) base_se)
               s1 (s, ^ev UndefinedExpr se)``],
       side_conditions = [],
       (* ---------------------------------------------------------- *)
       conclusion = (--`^mng (mVarD (VDecInit t n e)) s0 (s, Undefined)`--)},
      {hypotheses = [], side_conditions = [
         ``newstrmap = override (strmap s) name (str_info flds)``
       ],
       (* ---------------------------------------------------- *)
       conclusion = ``^mng (mVarD (VStrDec name flds)) s
                          (strmap_update newstrmap s, VarDeclVal)``},
      s (--`^mng (mVarDl []) s (s, VarDeclVal)`--),

      {hypotheses = [(--`^mng (mVarD vhd) s0 (s, Undefined)`--)],
       side_conditions = [],
       conclusion = (--`^mng (mVarDl (CONS vhd vtl)) s0 (s, Undefined)`--)},

      {hypotheses = [
         (--`^mng (mVarD vhd) s0 (s1, VarDeclVal)`--),
         (--`^mng (mVarDl vtl) s1 (s2, v)`--)
       ], side_conditions = [],
       (* ---------------------------------------------- *)
       conclusion =
         (--`^mng (mVarDl (CONS vhd vtl)) s0 (s2, v)`--)}
    ]
  };
in
  val (meaning_thms, mng_induction, mng_cases) =
    IndDefLib.new_inductive_definition ind_definition;
  val meaning = CONJ meaning_thms mng_induction
end;
```

Appendix B

Theorems

Subject reduction The `expr_type` relation is satisfied if the last argument is the type of the third, the expression.

```
``!e0 se0 s0 e se s.
     meaning (mExpr e0 se0) s0 (s,ExprVal e se) ==>
     has_no_inactive_undefineds e0 ==>
     fnapps_safe e0 ==>
     !v t.
       expr_type (expr_type_comps s0) v (INL e0) (INL t) ==>
       has_no_undefineds e ==>
       expr_type (expr_type_comps s) v (INL e) (INL t)``
```

Type safety The `rec_expr_P` functional recursively applies a predicate over the entirety of a syntax tree and checks that it holds everywhere. Here we check to see that the expression does not contain any function calls.

```
``!e0 t1 s0 se0 v.
     expr_type (expr_type_comps s0) v (INL e0) t1 ==>
     rec_expr_P e0 (\e. !f args. ~(e = FnApp f args) /\
                                 ~(e = FnApp_sqpt f args)) ==>
     lval_safe e0 ==> ~(e0 = UndefinedExpr) ==>
     (!v t2. ~(e0 = ECompVal v t2)) ==>
     ?e se s.
        meaning (mExpr (RValreq e0) se0) s0 (s, ExprVal (RValreq e) se)``
```

Syntactic purity implies semantic purity The requirement that a state be *pure* requires it to have no pending side effects and to have an empty update map. These are both components that the Cholera model stores in the side effect record, so it is implemented by the `pure_se` predicate.

```
``(!e0 se0 s0 e se s.
      meaning (mTCExpr e0 se0) s0 (s, ExprVal e se) ==>
      syn_pure_expr e0 ==> pure_se se0 ==>
      pure_se se /\ (s = s0))``
```

$\sim_R$ is a "pseudo-bisimulation" for *synpure* expressions The "bisimulation" between states required that the states be identical up to variance in reference maps only. This is met by requiring that the side effect records be pure, and that the initial states (s0) be the same.

```
``(!e0 se0 s0 e se s.
      meaning (mTCExpr e0 se0) s0 (s, ExprVal e se) ==>
      pure_se se0 ==> syn_pure_expr e0 ==>
      !se0'. pure_se se0' ==>
             ?se'. meaning (mTCExpr e0 se0') s0 (s0, ExprVal e se') /\
                   pure_se se')``
```

Single step diamond result for *synpure* expressions

```
``!e0 se0 s0 e1 se1 s1 e2 se2 s2 v t.
     meaning (mExpr e0 se0) s0 (s1, ExprVal e1 se1) /\
     meaning (mExpr e0 se0) s0 (s2, ExprVal e2 se2) /\
     pure_se se0 /\ syn_pure_expr e0 /\ has_no_undefineds e0 /\
     expr_type (expr_type_comps s0) v (INL e0) (INL t) /\
     ~((s1, e1, se1) = (s2, e2, se2)) ==>
     ?se1' se2' e.
        meaning (mExpr e1 se1) s1 (s0, ExprVal e se1') /\
        meaning (mExpr e2 se2) s2 (s0, ExprVal e se2') /\
        pure_se se1' /\ pure_se se2' /\
        (has_no_undefineds e =
          has_no_undefineds e1 /\ has_no_undefineds e2)``
```

Determinism of *synpure* expressions

```
``!e0 se0 s0 v1 t1 v2 t2 se1 se2 s1 s2.
    meaning (mTCExpr e0 se0) s0 (s1, ExprVal (ECompVal v1 t1) se1) /\
    meaning (mTCExpr e0 se0) s0 (s2, ExprVal (ECompVal v2 t2) se2) /\
    pure_se se0 /\ syn_pure_expr e0 /\ fnapps_safe e0 /\
    has_no_inactive_undefineds e0 /\
    (?t. expr_type (expr_type_comps s0) RValue (INL e0) (INL t)) ==>
    (s1 = s0) /\ (s2 = s0) /\ pure_se se1 /\ pure_se se2 /\
    (v1 = v2) /\ (t1 = t2)``
```

Pure reductions commute with apply_se This lemma, the first of those connected with the sequence point free determinism result consists of the statement of the result for both $\rightarrow_e$ and $\rightarrow_e^*$. We express the idea of being a "pure reduction" by holding the update_map constant before and after, and also requiring that any pending side effects not be "null" i.e., they must all affect at least a byte of memory. The rbag_safe and refbags constraints relate to properties that must hold of the bags of references maintained as part of the assignment syntax. Note also that in the mechanisation, apply_se is a relation that asserts that *a* side effect has been applied, but without specifying which one.

```
``(!e0 se' s' e se s.
     meaning (mTCExpr e0 se') s' (s, ExprVal e se) ==>
     seqpt_free e0 ==> rec_expr_P e0 rbag_safe ==>
     SUB_BAG (refbags e0) (ref_map se') ==>
     (?v t. expr_type (expr_type_comps s') v (INL e0) (INL t)) ==>
```

```
 (update_map se' = update_map se) ==> null_ise_free se' ==>
 has_no_undefineds e ==> safe_se (se,s) ==>
 !se0 s0. apply_se (se0, s0) (se', s') ==>
 ?se''. meaning (mTCExpr e0 se0) s0 (s0, ExprVal e se'') /\
        (update_map se0 = update_map se'') /\
        apply_se (se'', s0) (se, s))``
```

Separation The apply_nse relation relates two states if it is possible to get from one to the other by applying n side effects.

```
``(!e0 se0 s0 e se s.
     meaning (mTCExpr e0 se0) s0 (s, ExprVal e se) ==>
     seqpt_free e0 ==> rec_expr_P e0 rbag_safe ==>
     (?v t. expr_type (expr_type_comps s0) v (INL e0) (INL t)) ==>
     null_ise_free se0 ==> has_no_undefineds e ==>
     SUB_BAG (refbags e0) (ref_map se0) ==> safe_se (se, s) ==>
     ?se'.
         meaning (mTCExpr e0 se0) s0 (s0, ExprVal e se') /\
         (update_map se0 = update_map se') /\
         ?n. apply_nse n (se', s0) (se, s))``,
```

Reduction characterisation All sequence point free reductions that don't update the state can be characterised in terms of the way in which they affect the side effect record (the reference and update maps, and the bag of pending side effects). The following theorem is the statement of how such a reduction can be characterised. As well as the three state components mentioned above, we also have to characterise the refbags of the expression because these have an effect on the reference map when an assignment expression completes. The final part of the conclusion states that analogous reductions are possible from a variety of similar starting states. This will then allow the proof of the final diamond property.

```
``!e0 se0 s0 e se s.
    meaning (mTCExpr e0 se0) s0 (s, ExprVal e se) ==>
    seqpt_free e0 ==> null_ise_free se0 ==> has_no_undefineds e ==>
    SUB_BAG (refbags e0) (ref_map se0) ==> rec_expr_P e0 rbag_safe ==>
    (?t v. expr_type (expr_type_comps s0) v (INL e0) (INL t)) ==>
    (update_map se0 = update_map se) ==>
    rec_expr_P e rbag_safe /\ SUB_BAG (refbags e) (ref_map se) /\
    ?rbu rbd pb rbbu rbbd.
      SUB_BAG rbd rbbd /\ SUB_BAG rbbu rbu /\
      SUB_BAG rbbd (BAG_UNION (refbags e0) rbbu) /\
      (ref_map se = BAG_DIFF (BAG_UNION (ref_map se0) rbu) rbd) /\
      (refbags e = BAG_DIFF (BAG_UNION (refbags e0) rbbu) rbbd) /\
      (pending_ses se = BAG_UNION (pending_ses se0) pb) /\
      (!se0'. (update_map se0' = update_map se0) ==>
              SUB_BAG (refbags e0) (ref_map se0') ==>
       ?se'.
              meaning (mTCExpr e0 se0') s0 (s0, ExprVal e se') /\
              (update_map se' = update_map se0') /\
              (ref_map se' =
```

```
                    BAG_DIFF (BAG_UNION (ref_map se0') rbu) rbd) /\
                  (pending_ses se' = BAG_UNION (pending_ses se0') pb))``
```

Reduction preconditions preserved This theorem is actually embodied in a number of separate results across all of the Cholera theory files. The preconditions are all apparent from the *Reduction characterisation* result above. Some of the preservation sub-results are also presented there (the requirement that the expression `e0` be recursively `rbag_safe` is one such), while others are given their own separate theorem. An example we have already seen is the preservation of the typedness condition (subject reduction), while another (more trivial) example is the preservation of the `null_ise_free` condition, as follows (it is typical that this preservation property actually requires other preconditions in place as well):

```
``!e0 se0 s0 e se s.
     meaning (mTCExpr e0 se0) s0 (s, ExprVal e se) ==>
     seqpt_free e0 ==> has_no_undefineds e ==>
     (?t v. expr_type (expr_type_comps s0) v (INL e0) (INL t)) ==>
     null_ise_free se0 ==> null_ise_free se)``
```

Finite and unsafe states can become undefined This theorem appears much the same in the Cholera presentation as it does in chapter four.

```
``!e0 se0 s0.
     FINITE_BAG (pending_ses se0) ==>
     ~safe_se (se0,s0) ==>
     ?se s.
        meaning (mTCExpr e0 se0) s0 (s, ExprVal UndefinedExpr se)``
```

Undefined sub-expression can always ascend While the result is true of sequence point free expressions, in the Cholera development we actually prove the slightly stronger result that it is true as long as the expression has no inactive undefined sub-expressions, and is also `fnapps_safe`.

```
``!e. fnapps_safe e /\ ~has_no_undefineds e /\
      has_no_inactive_undefineds e ==>
      !s0 se0.
        ?s se. meaning (mTCExpr e se0) s0
                       (s, ExprVal UndefinedExpr se)``
```

The cliff's edge The "state" `(e'', se'', s'')` that is proved to exist in this lemma is effectively undefined, and therefore must admit a reduction sequence to full undefinedness, as long as the bag of pending side effects is finite.

```
``!e0 se0 s0 e se s v t.
    meaning (mExpr e0 se0) s0 (s, ExprVal e se) ==>
    expr_type (expr_type_comps s0) v (INL e0) (INL t) ==>
    has_no_undefineds e0 ==> safe_se (se0,s0) ==>
    seqpt_free e0 ==>
      (~has_no_undefineds e \/ ~safe_se (se,s)) ==>
      !e' se' s'.
        meaning (mExpr e0 se0) s0 (s', ExprVal e' se') ==>
```

```
has_no_undefineds e' ==> safe_se (se',s') ==>
?e'' se'' s''.
  meaning (mTCExpr e' se') s' (s'', ExprVal e'' se'') /\
  (~has_no_undefineds e'' \/ ~safe_se (se'',s''))``
```

A diamond property for $\rightarrow_E$ and $\rightarrow_A$ This result is expressed in terms of the `augmng_transition` relation, which is $\rightarrow_e$ augmented with all of the necessary preconditions.

```
``!e0 se0 s0 e1 se1 se2 s2.
     augmng_transition (e0, se0, s0) (e1, se1, s0) /\
     safe_se (se1,s0) /\ apply_se (se0, s0) (se2, s2) ==>
     ?se.
        augmng_transition (e0, se2, s2) (e1, se, s2) /\
        apply_se (se1, s0) (se, s2)``
```

The C invariant rule The `invariant` relation states that the invariant `i` is preserved as long as the starting state shares the same environment as the state `s`.

```
``!s i G bdy.
    invariant s i (mStmt (CIf G bdy EmptyStmt)) ==>
    invariant s i (mStmt (CLoop G bdy))``
```

Correctness of findint function The *findint* function is mechanised as `pstmt` ("process statement"), where the last boolean argument is true if the interrupt statement being sought is `return` and false if it is `break`. Further, `bexp_meaning g v s0 s` means that g evaluates with boolean value `v`, and alters starting state `s0` to `s` in the process.

```
``!g bdy s0 s v.
    meaning (mStmt (whileloop g bdy)) s0 (s, v) ==>
    (v = Undefined) \/
    ((v = StmtVal) /\ pstmt bdy [TrueG g] s T) \/
    ((v = StmtVal) /\ ?s'. bexp_meaning g F s' s) \/
    (?mv. (v = RetVal mv) /\ pstmt bdy [TrueG g] s F)``
```

Bibliography

[AA79] Michael A. Arbib and Suad Alagić. Proof rules for gotos. *Acta Informatica*, 11(2):139–148, 1979.

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, New York, 1996.

[And94] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[ANS89] American National Standards Institute, 1430 Broadway, New York, NY 10018, USA. *American National Standard Programming Language C, ANSI X3.159-1989*, December 1989.

[BK83] R.-J. R. Back and M. Karttunen. A predicate transformer semantics for statements with multiple exits. University of Helsinki, unpublished MS, 1983.

[Boe85] Hans-Juergen Boehm. Side effects and aliasing can have simple axiomatic descriptions. *ACM Transactions on Programming Languages and Systems*, 7(4):637–655, October 1985.

[Bof98] Mark Bofinger. *Reasoning about C programs*. PhD thesis, University of Queensland, February 1998.

[BRTT93] Lars Birkedal, Nick Rothwell, Mads Tofte, and David N. Turner. The ML kit (version 1). Technical Report DIKU-report 93/14, Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, 1993.

[Bry92] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[BW96] Paul Black and Phil Windley. Inference rules for programming languages with side effects. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem proving in higher order logics. 9th international conference, TPHOLs 96*, volume 1125 of *Lecture notes in computer science*, pages 51–60. Springer, August 1996.

[BW98] Paul E. Black and Phillip J. Windley. Formal verification of secure programs in the presence of side effects. In *Proceedings of the Thirty-first Hawai'i International Conference on System Sciences (HICSS-31)*, January 1998.

[Car97] Luca Cardelli. Type systems. In Allen B. Tucker, Jr., editor, *The Computer Science and Engineering Handbook*. CRC Press, 1997.

[CCR94] J. V. Cook, E. L. Cohen, and T. S. Redmond. A formal denotational semantics for C. A draft document, once available from Trusted Information Systems' web-site at `http://www.tis.com/docs/research/assurance/formal-c.html`, but now seemingly unavailable, September 1994.

[DE97] Sophia Drossopolou and Susan Eisenbach. Java is type safe—probably. In *11th European Conference on Object Oriented Programming*, number 1241 in Lecture Notes in Computer Science. Springer, June 1997.

[Dij76] Edsger W. Dijkstra. *A discipline of programming*. Prentice-Hall, 1976.

[EGHT94] David Evans, John V. Guttag, James J. Horning, and Yang Meng Tan. LCLint: a tool for using specifications to check code. In *SIGSOFT Symposium on the Foundations of Software Engineering*, December 1994.

[Eva96] David Evans. Static detection of dynamic memory errors. In *SIGPLAN Conference on Programming Language Design and Implementation*, Philadelphia, PA., May 1996.

[Fea98] Clive D. W. Feather. A theory of sequence points. ISO working paper ISO/IEC/JTC1/SC22/WG14/N822, 1998. Under consideration for inclusion as an Informative Annexe in the next version of the ISO C standard.

[FF86] M. Felleisen and D. Friedman. Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts III*, pages 193–217. North-Holland, 1986.

[GH93] Yuri Gurevich and James K. Huggins. The semantics of the C programming language. In E. Borger, editor, *Selected papers from CSL '92*, volume 702 of *Lecture notes in computer science*, pages 274–308. Spring-Verlag, 1993. Corrected version available from University of Michigan web-site: `http://www.eecs.umich.edu/gasm`.

[GM93] M. J. C. Gordon and T. Melham (editors). *Introduction to HOL: a theorem proving environment*. Cambridge University Press, 1993.

[GMW79] M. J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*. Number 78 in Lecture Notes in Computer Science. Springer, 1979.

[Gor89] M. J. C. Gordon. Mechanizing programming logics in higher order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current trends in hardware verification and automated theorem proving*. Springer-Verlag, 1989.

[Gor95] Andrew D. Gordon. Bisimilarity as a theory of functional programming. Mini-course. BRICS Notes Series NS–95–3, BRICS, Aarhus University, 1995. Extended version of MFPS'95 and Glasgow FP'94 papers.

[Gun93] E. L. Gunter. A broader class of trees for recursive type definitions for HOL. In J. J. Joyce and C.-J. H. Seger, editors, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 780 of *Lecture Notes in Computer Science*, pages 140–154, Vancouver, Canada, August 1993. University of British Columbia, Springer-Verlag, published 1994.

[Gur91] Yuri Gurevich. Evolving algebras: a tutorial introduction. *Bulletin of EATCS*, 43:264–284, 1991.

[Har95a] John Harrison. Binary decision diagrams as a HOL derived rule. *Computer Journal*, 38(2), 1995.

[Har95b] John Harrison. Inductive definitions: automation and application. In E. Thomas Schubert, Phillip J. Windley, and James

Alves-Foss, editors, *Higher order logic theorem proving and its applications. 8th international workshop*, volume 971 of *Lecture notes in computer science*, pages 200–213. Springer, September 1995.

[HM94] P.V. Homeier and D.F. Martin. Trustworthy tools for trustworthy programs: a verified verification condition generator. In T.F. Melham and J. Camilleri, editors, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 859 of *Lecture Notes in Computer Science*, pages 269–284, Valletta, Malta, September 1994. Springer-Verlag.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[Hut90] Matthew Hutchins. Machine assisted reasoning about standard ML using HOL, November 1990. Australian National University. Honours thesis.

[ISO90] *Programming languages – C*, 1990. ISO/IEC 9899:1990.

[JK95] Richard Jones and Paul Kelly. Bounds checking for C. Available from `http://www-ala.doc.ic.ac.uk/~phjk/BoundsChecking.html`, July 1995.

[JTC] ISO committee JTC1/SC22/WG14. Record of responses. Available from `ftp://ftp.dmk.com/DMK/sc22wg14/rr/`.

[Kah93] Stefan Kahrs. Mistakes and ambiguities in the definition of Standard ML. LFCS Report ECS-LFCS-93-257, University of Edinburgh, April 1993. An update listing further errors can be found at `ftp://ftp.dcs.ed.ac.uk/pub/smk/SML/errors-new.ps.Z`.

[KM95] Steve King and Carroll Morgan. Exits in the refinement calculus. *Formal aspects of computing*, 7(1):54–76, 1995.

[Kow77] T. Kowaltowski. Axiomatic approach to side effects and general jumps. *Acta Informatica*, 7(4):357–60, 1977.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language*. Prentice Hall, 2nd edition, 1988.

[Mel91] T. Melham. A package for inductive relation definitions in HOL. In Myla Archer, Jeffrey J. Joyce, Karl N. Levitt, and Phillip J. Windley, editors, *Proceedings of the 1991 international workshop on the HOL theorem proving system and its applications*, pages 350–357. IEEE Computer Society Press, 1991.

[Mil89] Robin Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

[MLP79] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):271–280, May 1979.

[MT90] Robin Milner and Mads Tofte. *Commentary on Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

[MTH90] Robin Milner, Mads Tofte, and Robert W. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

[Nec97] George Necula. Proof-carrying code. In *Conference record of POPL '97: The 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, January 1997.

[NvO98] Tobias Nipkow and David von Oheimb. Java$_{\ell ight}$ is type-safe — definitely. In *25th principles of programming languages*. ACM Press, 1998. To appear.

[ORS92] S. Owre, J. Rushby, and N. Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, 1992.

[Rit93] D. M. Ritchie. The development of the C language. *ACM SIGPLAN Notices*, 28(3):201–208, March 1993.

[Rus93] John Rushby. Formal methods and the certification of critical systems. Technical Report CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, CA 94025, USA, November 1993.

[SC96] Sakthi Subramanian and J. V. Cook. Mechanical verification of C programs. In *First workshop on Formal Methods in Software Practice (FMSP '96)*. Association for Computing Machinery, January 1996.

[SV98] Geoffrey Smith and Dennis Volpano. A sound polymorphic type system for a dialect of C. *Science of Computer Programming*, 32(2–3), 1998. To appear.

[Sym93] Donald Syme. Reasoning with the formal definition of Standard ML in HOL. In *Higher order logic theorem proving and its applications: 6th International Workshop, HUG '93*, number 780 in Lecture notes in computer science, Vancouver, B.C., August 1993. Springer-Verlag.

[Sym97a] Donald Syme. DECLARE: a prototype declarative proof system for higher order logic. Technical Report 416, Computer Laboratory, University of Cambridge, 1997.

[Sym97b] Donald Syme. Proving Java type soundness. Technical Report 427, Computer Laboratory, Univeristy of Cambridge, June 1997.

[Ten91] R. D. Tennent. *Semantics of programming languages*. Prentice Hall, 1991.

[Van93] Myra VanInwegen. HOL-ML. In *Higher order logic theorem proving and its applications: 6th International Workshop, HUG '93*, number 780 in Lecture notes in computer science, pages 59–72. Springer-Verlag, August 1993.

[Van96] Myra VanInwegen. *The machine-assisted proof of programming language properties*. PhD thesis, University of Pennsylvania, December 1996.

[Yu93] Yuan Yu. *Automated proofs of object code for a widely used microprocessor*. PhD thesis, University of Texas at Austin, April 1993.