Proving Correctness of Modular Functional Programs

Christopher Andrew Owens

Doctor of Philosophy
University of Edinburgh
1999

For Dad,
and for Mum.

I whacked the back of the driver's seat with my fist. "This is *important*, goddamnit! This is a *true story!*" The car swerved sickeningly, then straightened out. . . . The kid in the back looked like he was ready to jump right out of the car and take his chances.

Our vibrations were getting nasty—but why? I was puzzled, frustrated. Was there no communication in this car? Had we deteriorated to the level of *dumb beasts?*

Because my story *was* true. I was certain of that. And it was extremely important, I felt, for the *meaning* of our journey to be made absolutely clear. . . . And when the call came, I was ready.

—Hunter S. Thompson, *Fear and Loathing in Las Vegas*

Abstract

One reason for studying and programming in functional programming languages is that they are easy to reason about, yet there is surprisingly little work on proving the correctness of large functional programs. In this dissertation I show how to provide a system for proving the correctness of large programs written in a major functional programming language, ML [MTH90]. ML is split into two parts: the Core (in which the actual programs are written), and Modules (which are used to structure Core programs).

The dissertation has three main themes.

- Due to the detail and complexity of proofs of programs, a realistic system should use a computer proof assistant, and so I first discuss how such a system can be coded in a generic proof assistant (I use Paulson's Isabelle [Pau94a]).

- I give a formal proof system for proving properties of programs written in the functional part of Core ML.

- The Modules language is one of ML's strengths: it allows the programmer to write large programs by controlling the interactions between its parts. In the final part of the dissertation I give a method of proving correctness of ML Modules programs using the well-known data reification method [Jon86]. Proofs of reification using this method boil down to proofs in the system for the Core.

Acknowledgements

> This fiction is for Beamish, whom, while en route for some conference or other, I last saw at Frankfurt airport, enquiring from desk to desk about his luggage, unhappily not loaded onto the same plane as he. It is a total invention with delusory approximations to historical reality, just as is history itself. Not only does the University of Watermouth, which appears here, bear no relation to the real University of Watermouth (which does not exist) or to any other university; the year 1972, which also appears, bears no relation to the real 1972, which was a fiction anyway, and so on. As for the characters, so-called, no one but the other characters in this book knows them, and they not very well; they are pure inventions, as is the plot in which they more than participate. Nor did I fly to a conference the other day; and if I did, there was no one on the plane named Beamish, who certainly did not lose his luggage. The rest, of course, is true.
>
> —Malcolm Bradbury, *The History Man*

This dissertation could not have been written without the help of my supervisors, Stephen Gilmore and Stuart Anderson. Stephen listened to every single one of my academic problems for four years, for which he deserves a Purple Heart. Stuart was always incisive and gave me a sense of urgency (although you'd never know it). I will particularly miss our supervisions. Thanks to them I can say, with barely a hint of irony, that Computer Science is a strange and beautiful subject.

Other members of the LFCS in Edinburgh have given assistance: James McKinna is always willing to converse about life, computer science in general, and my work in particular. Mark Jerrum and Colin Stirling helped me out of a tricky situation. Rod Burstall gave me a temporary job and taught me that I am a grain of sand. The members of the Specifications Club have put up with talks on various incorrect versions of my work.

In Cambridge, I have, of course, to thank Larry Paulson, Mike Gordon and Andy Gordon for giving me a job, and for not minding too much that I was a terrible research assistant. My colleagues in the Automated Reasoning Group made me feel welcome, and rarely objected to my stupid questions.

Out in the real world, I'm grateful to Jonathan Flowers, Andy South, Margaret Cullum, David Head, Karen Meakings and Julie Whitworth (amongst others) at NatWest Consultancy, some people at The Technology Broker, and especially everyone at Scientia.

Various officemates, flatmates and friends have put up with a lot: Steve Tweedie, James McKinna (again), Jennifer Martin, Keri Torney, Neil "drinkies?" Ghani, Christoph Lüth, John Longley, Dilip Sequeira, Tim Heap, Alison Jones, Kirsteen Bruce, Simon (and Laura) Blackwell, Jeni Haste, Matt Shrimpton, Moira Chalmers, Don Syme, Michael Norrish, Mark Staples, Myra VanInwegen, Chae Henson, Suzanne Slaughter and others too numerous to mention. Special mention to Dave Aspinall, Neil Ghani (again) and James McKinna (again again), who managed to massage down innumerable pints in order to keep me company.

Various other people have helped: Kristin Hersh, Thurston Moore and Kim Gordon, and Arsène Wenger. Craig McDevitt helped me learn some things about buddha-nature.

Thanks to Mum for everything, but in particular her support (financial and otherwise) over the last seven years. Thanks to Dave, my brother, for advice and computer games. Lastly, thanks to Eleanor, for everything else.

Declaration

I declare that this thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text.

(*Christopher Andrew Owens*)

Table of Contents

Chapter 1

Introduction

Words are trains
For moving past what really has no name
—Paddy MacAloon, *Couldn't Bear To Be Special*

Whereof one cannot speak thereof one must be silent.
—Ludwig Wittgenstein, *Tractatus Logico-Philosophicus*

1.1 Hypothesis

This dissertation will show the following.

- The encoding of logics for programs in logical frameworks can be made clearer and more efficient.
- Programs in a subset of Standard ML are amenable to proofs using such a logic.
- A variation of data reification can be used to prove Standard ML programs correct.

1.2 Related work

Related work is discussed in the relevant chapter.

1.3 Results

- I give a use and an evaluation of McKinna-Pollack binding.
- I give a new way of coding judgements in logical frameworks that reduces the amount of time spent checking well-formedness conditions during proof.

- I give a logic for reasoning about a significant part of (Core) ML.
- I define data reification for (Modules) ML.
- I suggest a proof that reification is transitive. This is a significant advance over previous systems for languages with higher-order functions (specifically, Tennent's work). However, at present the only proof is in a classical logic and is non-constructive: since it gives no way to construct as ML expressions the values it claims exists, it can only be regarded as an indication of the result, rather than a full proof.

1.4 Proofs, social processes, formal methods and critical systems

It is too often the case that program verification work is unmotivated except for the briefest of references to "the software crisis". Whilst beautiful Computer Science needs no further motivation, it is my aim to place my work in context and to assess its possible uses.

1.4.1 What is a proof?

There is clearly a difference between informal (but rigorous) and formal proof. Informal proof—and its cycle of "proofs" and "refutations"—is discussed by Lakatos [Lak76]. In an (in)famous paper, De Millo, Lipton and Perlis [MLP79] incorrectly apply these ideas to formal, computer-aided proof. In essence, they claim that, since computer proofs are too long, too detailed, and too dull to be read by humans, they can never be subjected to the analysis and the social processes which give us confidence that proofs are correct.

As Pollack [Pol96] observes, there *is* (or could be) a social process at work which can give us confidence in computer-aided proof, but it does not involve direct scrutiny of the proof by human mathematicians.

To be fair to De Millo, Lipton and Perlis, things have moved on since the mid-70s—it is no longer considered the cutting edge of proof technology to have a huge, ad hoc program written for a single proof which thinks deeply for some period of time before announcing an inscrutable "QED", and they are quite right to attack this sort of proof.

What, then, is a formal proof? First, note that there is an informal proof underlying all but the simplest formal proofs. As anyone who has used a theorem prover knows, the most useful tool for use with an automatic theorem prover is a blackboard. The correctness of the informal proof is obviously tied up with the correctness of the formal proof—the informal proof gains credibility if the formal proof is correct, but also the formal proof gains credibility if the informal proof is understood.

The formality of formal proof lies in the manipulation of the symbols in the proof. Let us assume that the logic in which the proofs are conducted is widely accepted (via proofs which are accepted via the social process) to be sound. The validity of the proof relies, in theory, simply on the absence of syntactic errors in the application of rules. How can we assure ourselves of the absence of these errors?

Automated theorem provers (supposedly) eliminate the possibility of error in formal proof. De Millo, Lipton and Perlis claim that automated proofs (especially proofs of programs) cannot be subject to the social processes that ordinary proofs are, and are thus worthless.

In fact, the situation is more complex. Automated theorem provers could certainly contain programming errors which allow false results to be proved. It would certainly be foolhardy to believe a result simply because it was proved in a theorem prover. However, many theorem provers (HOL [GM93] is the shining example) have large bodies of results which they have proved. These proofs, when published, can allow the social process to be applied to the prover itself, and we can thus gain some measure of confidence in the correctness of the prover, by virtue of the fact that we believe it has never proved anything which we believe false. Conversely, of course, confidence in the prover increases confidence in the theorems it proves.

Bundy [Bun91] calls for a "science of reasoning", which could be applied to automated theorem proving. The proof scripts which would result would reflect the informal proof—concentrating on the structure of the proof and important details, while ignoring unimportant details. We are certainly a long way from a time when proof scripts for most theorem provers fulfil this criterion. However, where they do, they too are subject to the social process. They give a summary of the informal proof, and indicate that the formal proof follows the informal.

It appears that documents such as DEF STAN 00-55, which set standards for correctness of programs, will soon require that theorem provers are capable of producing a trace of every atomic proof step. Such a proof could then be checked by a proof checker, which can be very much simpler than the prover itself. Such checkers might be sufficiently simple to be proved correct. Although this is not a guarantee that they would work as one might wish (as discussed later), it certainly adds confidence to the checked theorems. It is to be regretted that at the moment there seem to be few such checkers.

Similarly, theorem provers based on constructive type theory, such as Pollack's LEGO [PL92], can exhibit proof terms. In order to show that the proof is correct, it is only necessary to show that the proof term has the type claimed for it. Such type checkers can be extremely simple, and one might even imagine not only that there would be one or two verified proof checkers, but that every user could write their own.

1.4.2 What does "verified" mean?

Suppose that we prove a program correct. We have taken some mathematical model of the program and proven that it meets some mathematical specification. Leaving aside the correctness of the proofs, there are two problems:

- the specification may not capture the behaviour we intended;
- the actual program behaviour may not accord with our specification.

Essentially, there is no way to ensure that a specification correctly captures the program behaviour we intend: our intentions are in our head, and we must transcribe them to paper. There is no way that we can be sure that we have transcribed them correctly. We might test a specification by various thought experiments, by proving desirable properties of the specification. This can give us a measure of confidence, but it cannot ensure that we have not omitted some vital part of the specification which we have omitted to test. This is the "specification gap".

Now consider the second problem. Few, if any, compilers are proven to compile correctly. Even if they do, few processors have been proven correct, and even fewer computers have been proven to be correct. Suppose that we are running a trusted compiler on a trusted processor design which is part of trusted computer design: this does not eliminate the possibility of fabrication errors in the computer or processor, or that the computer or chip may simply wear out and become unreliable, or that the computer's memory is struck by a cosmic ray. Cohn [Coh89] gives a useful introduction to these problems in the context of hardware verification.

1.4.3 Is ML appropriate for critical systems?

In short, and in general: no. It must be possible to ensure that a critical program never runs out of memory. This means that languages for critical systems (for example SPADE Ada) can have their memory use determined statically at compile time. This in turn means that they do not allocate fresh memory (as C's `malloc` or Pascal's `new` do). Since ML is a memory-managed language where a program has no way to control or even observe how much memory it uses, either on the heap or on the call stack, it is all too easy to write programs which run out of space and so abort.

1.4.4 What critical systems might use ML?

Despite the fact that we cannot guarantee that a non-trivial ML program will not run out of memory, and despite the fact that ML is far from simple to compile, there are critical applications where the use of ML is justified.

Let us look at an example. Gordon [Gor94] briefly describes a system in which a computer screens cervical smears for abnormal cells (which was, in fact, being programmed in the functional language Haskell). The advantage of using a functional language here is that the task is conceptually hard, and a high-level language makes it easier (faster, less prone to error) to capture the designer's intentions. It is clearly critical that the computer gives no false negatives. It is not, however, critical that the computer does run out of memory—if it does, the sample can either be processed again by a computer with more memory or processed by hand.

What is it about this example which means that functional languages can be used? We might characterize it as "high-level, critical data-processing." We require some correctness properties—if the computer does not crash it provides the correct answer (no false negatives in the example). Since the problem is a hard one, the gains of using a high-level language outweigh the disadvantages such as the possibility of compiler bugs. The computation is, however, "off-line" in the sense that in the event of a crash the computation can simply be re-done.

There is one more aspect to programming in ML which makes it useful in critical applications: the Modules system. Let us consider what Perrow [Per84] calls "system" (or "normal") accidents. Such accidents occur in systems which have two features:

- They are closely coupled, meaning that each part depends on many other parts.
- Their parts have complex linking, meaning that there is not a single "thread" of control.

Together, these features mean that several independent failures can interact to cause a catastrophic failure. Perrow describes the failures in the Three Mile Island nuclear accident: there were several apparently unrelated failures. Taken separately, none of these was particularly serious. Together, they caused the most serious nuclear accident up to that time. Perrow argues that, since minor failures occur regularly, there will eventually come a time when several minor failures interact to cause a system accident. There are two ways to avoid this:

- reduce the complexity of the system (often at a cost to system performance) or reduce the coupling of the parts of the system (which may entail re-engineering the system);

- when complexity and close coupling are unavoidable, and the costs of a system accident are unacceptable, do not build the system.

The ML Modules system gives us a way to document and control the coupling between separate parts of the system. Using it for large programming tasks means that we can systematically control coupling, and thus reduce the risk of system accidents.

1.4.5 What have I missed?

In this section, I have only scratched the surface of a huge subject. Many issues have not been addressed, for example:

- what is a critical system?

- what is the environment of a critical system?

- what can program verification ensure?

- what is a safety case?

- what parts of critical systems need to be verified?

- can we achieve sufficient dependability in critical systems?

1.5 Structure of the dissertation

The main body of the dissertation falls roughly into three parts, in addition to which there is this introduction, a chapter of mathematical preliminaries, a conclusion and an appendix.

Background: theorem proving and formal methods: we give a brief historical review of the background to this dissertation in automated theorem proving and formal methods.

Notation, symbols and mathematical preliminaries: aside from some basic mathematical machinery, this chapter contains several results about partial equivalence relations which are used in the *properties of reification* chapter.

Part I: Logical Frameworks

Coding binding and substitution explicitly in Isabelle: in this chapter we describe an example logic coded in Isabelle using McKinna-Pollack binding [MP93]. Logics in later chapters will use this binding system. The Isabelle source code for the logic described in this chapter is given in an appendix.

Coding logics in logical frameworks: this chapter continues the discussion of how to code logics for programs in a logical framework. It outlines two techniques; these techniques are illustrated by the logic given in the chapter *reasoning about Core Simplified ML.*

Part II: Core ML

Simplified ML: in this chapter we specify the programming language Simplified ML. The remainder of the dissertation is concerned with reasoning about this language. We specify the Core language in some detail, and specify some key properties of the Modules language.

Reasoning about Core Simplified ML: we are now in a position to give a logic for reasoning about Core Simplified ML programs. This logic can be used to prove the correctness obligations to which the methods of the next chapter give rise.

Part III: Data Reification

Data reification: in this chapter we show how to prove ML Modules programs correct with respect to a specification, using the data reification method.

An investigation of the properties of reification: we investigate a variety of properties of the reification method described in the previous chapter, culminating in the conjecture that reification is transitive: that is, we can use step-wise refinement to write programs.

Case studies of data reification: we give a concrete illustration of the process of program development using the techniques we have described. We give a specification, reify it to give a first implementation, and reify that implementation to give a second implementation.

Conclusion: we give an overview of the dissertation, and describe how the work in it can be taken forward.

Chapter 2

Background: theorem proving and formal methods

> Ignorance is the first requisite of the historian.
>
> —Lytton Strachey, *Eminent Victorians*

This chapter is a survey of previous work in mechanized theorem proving and formal methods. It mixes historical, chronological treatment with analytical. Furthermore, and most importantly, no survey as brief as this could hope to be comprehensive: I attempt to cover what I regard as the main trends, but this selection is of course personal and partial. Although theorem proving and formal methods are tightly intertwined, we shall treat them separately in this chapter—despite the fact that this is an artificial distinction.

A different, more accomplished, historical summary is given by Jones [Jon92], and (separately) an extensive bibliography is given [JM92]. These works focus primarily on formal methods.

2.1 Formal methods

The study of formal methods would have been impossible before the twentieth century. It was only in the late nineteenth century and early twentieth century that logicians such as Peano, Hilbert, Frege and Russell formalized the basis of mathematics. "Formalized" in the sense that they reduced mathematics to a small number of axioms and laws which can be applied entirely mechanically by manipulating symbols (for instance by a computer). We shall skim over this early philosophical beginning—and indeed the mathematical philosophy which was a necessary prelude to it—and mention only Russell and Whitehead's *Principia Mathematica* [WR10], which remains influential to this day.

The study of formal methods then moves forward to the nineteen thirties when the solution of Hilbert's Entscheidungs Problem, by Church [Chu65b, Chu65a] and Turing [Tur36], gave us a mathematical model of computation. This mathematical model appeared some years before physical computers had been built and these pioneering computers could only be constructed once the mathematical foundations had been put in place. Although Church's work on the λ-calculus and Turing's on Turing machines contained the first proofs about computers and the properties of their programs this was not yet formal methods as we know it. Formal methods is an attempt to set forward methodologies to prove the correctness of computer programs using logic. However, Church and Turing both proved properties of particular computers with particular (universal) programs.

The first work which could reasonably be called formal methods was Turing's *Checking a Large Routine* [Tur49] (this has been published in the *Annals of Computing History*, edited by Morris and Jones [MJ84], who have included some commentary on the paper, its significance and, indeed, point out some minor errors in his proofs). In this paper Turing uses pre- and post-conditions in a way which prefigures their use in Floyd-Hoare logics. In retrospect, it is easy to recognize the importance of this paper; at the time it remained obscure, and little or no work followed directly from it.

We now turn to the subject of Floyd-Hoare logics, originally proposed by Floyd [Flo67] as a logic for reasoning about flowcharts, and later refined to the form which is now familiar by Hoare [Hoa69]. Floyd-Hoare logics remain one of the most popular ways of reasoning about imperative programs. The basic judgements in Floyd-Hoare logics are triples of the form $\{P\}C\{Q\}$, where P is the *pre-condition*, C is the command or program fragment to be executed and Q is the *post-condition*. The meaning of the Floyd-Hoare triple is: supposing P holds, then, after executing command C, Q will hold. A recent treatment of Floyd-Hoare logics in a theorem prover is given by Thomas Kleymann [Kle98]. In this form Floyd-Hoare logics only establish *partial correctness*: if the program fragment C terminates it satisfies the Hoare triple. It does not necessarily mean that a program terminates: consider, for example, a typical deduction rule for `while` loops.

$$\frac{\{P\}C\{P\}}{\{P\}\mathtt{while}\ B\ \mathtt{do}\ C\ \mathtt{end}\{P \wedge \neg B\}}$$

This rule does not say that the loop condition B definitely will become false, only that if the loop terminates it will be because B has become false. In order to establish that the loop does terminate one typically adds a measure function on a well-founded ordering which one then demonstrates is reducing, as shown in the rule below.

$$\frac{[P \wedge f(S) = x]C[P \wedge f(S') < x]}{[P]\texttt{while } B \texttt{ do } C \texttt{ end}[P \wedge \neg B]}$$

Notice that the triple is now written using square brackets, [_]_[_], to indicate that it is a judgement about total correctness (partial correctness plus termination). The function f takes as an argument the program state, S: often it will just be a statement about one or two program variables. Notice that we use S' in the post-condition: primed program variables indicate that we are considering the program state after the execution of C. The variable x is not a program variable, it is a variable of the logic, called an *auxiliary* variable.

Much modern formal methods work can be divided roughly into two camps: the *algebraic* in which program constructors are considered as operators in an algebra; and *model theoretic* where one first characterizes a single (often set-theoretic) model corresponding to a specification of the correct program and then refines this to give a single model which is the finished program.

In model theoretic methods, the original model is characterized by an abstract mathematical specification, and the process of turning this into a real, workable program in an executable programming language is *reification*: the process of turning an abstract idea into a concrete one. Examples of model theoretic methods are Z [ASM79], B [Abr96], and VDM [Jon73]. VDM is divided into a set theoretic specification language, a method for turning abstract data structures and operations on them into concrete data structures (data reification) and a method for turning abstract procedures into concrete procedures (procedure reification). In this thesis we will mostly neglect the process of procedure reification since data reification so neatly models the process of program development in a functional language, as we observe in Chapter 8. Data reification was originally proposed by Hoare [Hoa72]: we will discuss it in much greater length in Chapter 8.

We now come to algebraic specification, included in this are such work as OBJ [GW88], CLEAR [BG81] and ASL [SW83]. Of particular significance to the work in this thesis is Extended ML [KST94, San91]. The motivations for Extended ML are similar to the motivations for this thesis: to prove correctness of real ML programs. It differs in several respects however. The prime one of these is that it concerns itself with the whole of Standard ML, not merely the purely functional part, and includes exceptions and mutable store. This means that the definition of Extended ML is extremely complex: it runs to hundreds of pages and includes the definition of Standard ML as a proper subset. Given that there have been corrections and modifications made to the definition of Standard ML since it was written due to the study of many groups of academics, (some, for example, found due to work on Extended ML [Kah93]), it seems unlikely that the definition of Extended ML is yet entirely without error. (This is not of course to say that it is not a useful piece of work). Furthermore, the definition of Extended ML includes only its semantics: that is, it specifies only the models of the language and how the operators modify those models. It does not give any syntactic deduction rules for reasoning about Extended ML specifications. Given that extended ML includes the full power of standard ML—polymorphism, higher-order functions, exceptions, mutable state and modules—the task of producing such a set of deduction rules is formidable in the extreme.

One further difficulty with Extended ML is that it takes the ML type of booleans to be the type of propositions. This means that when defining or proving logical propositions one has the ability to include fragments of ML as one wishes, but it has the powerful disadvantage that truth and falsity are not the only values that propositions may hold: they may also fail to terminate. This adds significantly to the complexity of any deduction system: one must first show that a proposition denotes at all before reasoning about its truth.

Despite all these reservations however, Extended ML is the pre-eminent piece of work on algebraic specification of a real functional programming language. This thesis could not have been written without the pioneering work of Extended ML's inventors, definers and other workers.

2.2 Theorem proving

There are many different schools in automated theorem proving, some of which this short survey will necessarily ignore completely: for example model checking, perhaps best typified by SMV [McM92]. Many totally automated theorem provers such as Otter [Wos96] will also be ignored.

For the purposes of this dissertation the seminal work in theorem proving was done by Milner on his Stanford LCF system [Mil72]. Amongst its many innovations (including the invention of ML) perhaps the key one is that theorems are strongly typed: that is to say there is a type of theorems `thm` and only a very small number of operations on values of type `thm`. Since the language in which proofs are constructed is strongly typed, this means that one cannot produce values of type `thm` except using these functions. Assuming that these functions are indeed correct (that is, sound), we can then guarantee that all values of type `thm` are in fact theorems. Modern theorem provers such as HOL [GM93] are direct descendants of LCF.

Another descendant of LCF, although slightly less directly than HOL, is Paulson's Isabelle [Pau94a]. The defining characteristic of Isabelle is that it is generic: that is, one can define the rules of arbitrary logics within Isabelle and use it to prove theorems within those logics.

Another important strand in theorem proving is based on type theory: the Curry-Howard isomorphism tells us that types in λ-calculi relate to logical propositions, and values of those types can be regarded as proofs of those propositions. One can then use the type inference rules of a lambda calculus as the inference rules of a logic. A proof begins by hypothesizing that some type (proposition) is non-empty (has a proof), and continues by attempting to construct a member of the type. The correctness of the final proof is guaranteed by the fact that one can independently verify the type of the term which purports to be a proof of the theorem. Since type-checking is considerably simpler than theorem proving, this means that, as long as we have confidence in our type checker (which may or may not be part of the theorem prover), we can have confidence in the correctness of theorems proved by the theorem prover as a whole. This particular view is propounded by Pollack [Pol96].

Type theory necessarily gives us constructive logics (we are constructing proof terms as witnesses of our theorems). Type theory as a logic therefore has it roots in the work of Martin-Löf [ML75, ML80], Intuitionism [Dum77], and even Bishop's constructive analysis [Bis67].

The earliest important theorem prover using a type theory was de Bruijn's AUTOMATH project [dB80], followed by NuPrl [C^+86]. More recently λ-calculi have been used as systems within which logics can be defined, meaning that theorem provers based on certain type theories (the Edinburgh Logical Framework [HHP87] is one such theory) can be used as generic theorem provers, (thus, by a very different route, arriving in a similar place to the Isabelle system). Important modern type-theoretic theorem provers include Pollack's Lego [PL92] (based on Luo's ECC [Luo89] and which can be used as a reasoning tool for the Edinburgh logical framework); Coq [DFH^+93] (based on the similar Calculus of Constructions [CH86]); and ALF [AGNvS94]. These theorem provers now include such advanced tools as inductive data types and inversion of inductive relations.

Type theory lends itself particularly well to reasoning about programs, in that it allows a style of reasoning called program synthesis. In this style, one begins by postulating that there is a program with given properties. The process of proving this proposition results in the program itself being produced from the proof by Curry-Howard isomorphism. One therefore can imagine that the amount of work is halved: the program comes "for free" from the proof of its correctness. However, the most natural proof of a proposition may not result in the most effective algorithm. Take, for example, sorting: one might well imagine an inductive proof on a list of values which produced an algorithm such as selection sort, however it is difficult to imagine what proof would produce Hoare's Quicksort as a program. It is furthermore the case that values in a typed λ-calculus (which are the "programs" produced by this method) cannot capture the richness and expressiveness of programs in modern programming languages such as ML or Java.

Other important theorem provers not so far mentioned include the Boyer-Moore theorem provers Nqthm [BM79] and ACL2 (ACL2 is essentially a re-implementation of Nqthm). These have been used widely, for example in proving Gödel's incompleteness theorems [Sha94]. Much of the interest in the Boyer Moore theorem provers lies in the fact that it uses a weak logic without quantifiers. However, this enables the use of extremely powerful automatic reasoning methods, and, using various work-arounds such as Skolemization, it is possible to represent many of the theorems of more powerful logics in this restricted logic. Clam [BvHHS90] is a type theoretic prover based on the ideas of NuPrl. It is of significance to us because of the work that has been done on it in creating AI methods for automated mathematical proof, in particular rippling [BSvH$^+$93], a powerful method for inductive proof. Finally we mention SRI's PVS [OSR95], which is powerful and easy to use. This has lead to its widespread adoption as the theorem prover of choice for many formal methods projects, although its lack of programmability means that it is little used by researchers of theorem proving.

Chapter 3

Notation, symbols and mathematical preliminaries

> It is undesirable to believe a proposition when there is no ground whatever for supposing it is true. —Bertrand Russell

3.1 Logic

Most of the results in this dissertation are expressed in Higher Order Logic. This is a classical logic: in particular, a constructivist would object to the fact that it includes the law of the excluded middle, and that it contains a non-constructive description operator. This logic is used since it is the basis of the logic I give for reasoning about ML programs in Chapter 7.

I largely avoid relying on non-constructive proofs, since then the results can be translated to a constructive framework. However, on occasion I give a non-constructive proof either because I do not know of a constructive one, or because the constructive proof is complex and obscures the proof idea. The text notes when a proof is non-constructive.

The notation used is mainly standard, but the following points should be noted.

Truth values: logical truth is written T, and falsity is F. The type of truth values is written o.

Functions: function types in the logic are written $\tau \twoheadrightarrow \sigma$. (ML function types are written $\tau \rightarrow \sigma$). Application of a function in the logic is written in the same way as ML function application: $f\,a$.

Sets: as is usual in Higher Order Logic, we identify sets of values of type τ with their characteristic predicate $\tau \twoheadrightarrow o$.

Relations: similarly, n-ary relations between values of types $\tau_1, \ldots, \tau_n$ are predicates $\tau_1 \twoheadrightarrow \ldots \twoheadrightarrow \tau_n \twoheadrightarrow o$. If R is a binary relation, I shall usually write $x\ R\ y$ instead of Rxy to mean "x is related by R to y".

Implication: "A implies B" is written as $A \supset B$. As usual, $A \supset B \supset C$ associates as $A \supset (B \supset C)$.

Quantification and abstraction: Higher Order Logic is typed, and the type of quantified and λ abstracted variables is given explicitly. For example: $\forall x{:}\,\tau.\,P$.

Bounded universal quantification: we will often use bounded quantification, such as $\forall x{:}\,\tau|\,P.\,Q$. This is read as "for all x of type τ such that $P(x)$ holds, it is the case that Q holds". It is logically equivalent to $\forall x{:}\,\tau.\,P(x) \supset Q$.

Bounded existential quantification: $\exists x{:}\,\tau|\,P.\,Q$ is read as "there exists an x of type τ such that $P(x)$ holds, for which Q holds". It is equivalent to $\exists x{:}\,\tau.\,P(x) \wedge Q$.

Descriptions: Hilbert's "any" operator is written ε, and its use is typed: $\varepsilon x{:}\,\tau.\,P$. This is read as "any x of type τ such that $P(x)$ holds". If there is no x for which $P(x)$ holds, the choice is arbitrary.

The logic contains no empty types: this makes the treatment of quantifiers simpler ($\exists x{:}\,\tau.\,\mathsf{T}$ is always true without empty types), and also the treatment of descriptions (what does $\varepsilon x{:}\,\tau.\,P$ mean if τ has no elements?). Empty types in Higher Order Logic are discussed at greater length by Paulson [Pau90]. We also allow types to contain unknown type variables $\alpha, \beta, \ldots$, in the manner of the HOL logic [GM93]. The meaning of a judgement containing such polymorphic types is "this statement is true, no matter what monomorphic types are substituted for the type variables". It is not *parametric* polymorphism in the sense that Reynolds [Rey83] uses, since, for example, the judgement $\exists x{:}\,\alpha \twoheadrightarrow \beta.\,\mathsf{T}$ is valid. This is because, no matter what α and β are, the type is always inhabited. In a parametric λ calculus, there are no functions of type $\alpha \twoheadrightarrow \beta$.

3.2 Set theory

Most of the definitions in this section are standard.

Definition 3.1 *Let $R : \sigma \twoheadrightarrow \sigma \twoheadrightarrow o$ be a binary relation. The* opposite relation *R^{op} is given by:*

$$x\ R^{op}\ y \iff y\ R\ x$$

Definition 3.2 *Let $R : \sigma \twoheadrightarrow \sigma \twoheadrightarrow o$ be a binary relation:*

- *R is* reflexive *if and only if, for every x, $x\ R\ x$;*
- *R is* symmetric *if and only if, for every x and y, $x\ R\ y \supset y\ R\ x$;*
- *R is* transitive *if and only if, for every x, y and z, $x\ R\ y \supset y\ R\ z \supset x\ R\ z$.*

An equivalence relation *is reflexive, symmetric and transitive. A* partial equivalence relation*, or* per*, is symmetric and transitive.*

We shall discuss pers at greater length in the next section.

Definition 3.3 *Let $R : \sigma \twoheadrightarrow \tau \twoheadrightarrow o$ be a binary relation:*

- *R is* total *if and only if $\forall x{:}\,\sigma.\,\exists y{:}\,\tau.\,x\ R\ y$;*
- *R is* surjective *if and only if $\forall y{:}\,\tau.\,\exists x{:}\,\sigma.\,x\ R\ y$;*
- *R is* partial functional *if and only if $\forall x{:}\,\sigma.\,\forall y, y_1{:}\,\tau.\,x\ R\ y \supset x\ R\ y_1 \supset y = y_1$.*

These definitions are standard, except that we do not require R to be a partial function before we describe it as total or surjective. We say that R is total, surjective or partial functional from right to left *when R^{op} is total, surjective or partial functional respectively.*

Definition 3.4 *Given two relations $R : \tau_1 \twoheadrightarrow \tau_2 \twoheadrightarrow o$ and $S : \tau_2 \twoheadrightarrow \tau_3 \twoheadrightarrow o$, with a per $E : \tau_2 \twoheadrightarrow \tau_2 \twoheadrightarrow o$, and $W(x) = x\ E\ x$, then define $R \circ S$ as:*

$$x\ (R \circ S)\ z = \exists y{:}\,\tau_2.\,x\ R\ y \wedge y\ S\ z$$

3.3 Partial equivalences

Pers are not necessarily reflexive, but if E is a per, and there is a y such that $x\ E\ y$ or $y\ E\ x$, then $x\ E\ x$. Values for which $x\ E\ x$ will be referred to as *well-behaved* with respect to E. We will sometimes define $W(x) = x\ E\ x$.

It is useful to define some of the usual set theory terms for well-behaved values.

Definition 3.5 *Let $R : \sigma \twoheadrightarrow \tau \twoheadrightarrow o$ be a binary relation, and let $E_\sigma : \sigma \twoheadrightarrow \sigma \twoheadrightarrow o$ and $E_\tau : \tau \twoheadrightarrow \tau \twoheadrightarrow o$ be pers. Let $W_\sigma : \sigma \twoheadrightarrow o$ and $W_\tau : \tau \twoheadrightarrow o$ be well-behavedness predicates (in practice we will often define $W_\sigma(x) = x\ E_\sigma\ x$, and $W_\tau(y) = y\ E_\tau\ y$).*

- R *is* total up to (W_σ, W_τ) *(or* $(W_\sigma,\, W_\tau)$ total*) when:*

$$\forall x\colon \sigma |\, W_\sigma .\, \exists y\colon \tau |\, W_\tau .\, x \; R \; y$$

- R *is* surjective up to (W_σ, W_τ) *(or* $(W_\sigma,\, W_\tau)$ surjective*) when:*

$$\forall y\colon \tau |\, W_\tau .\, \exists x\colon \sigma |\, W_\sigma .\, x \; R \; y$$

Now we come to the key ideas of consistency and exactness. These are used extensively in chapter 8. Given a relation $R : \sigma \twoheadrightarrow \tau \twoheadrightarrow o$ and a per $E : \sigma \twoheadrightarrow \sigma \twoheadrightarrow o$, consistency says when R-relatedness can be deduced from E-equivalence, and exactness says when E-equivalence can be deduced from R-relatedness.

Definition 3.6 *Let $R : \sigma \twoheadrightarrow \tau \twoheadrightarrow o$ be a binary relation, let $E : \sigma \twoheadrightarrow \sigma \twoheadrightarrow o$ be a per.*

- R *is* left-consistent *with E (or E left-consistent) if and only if:*

$$x \; E \; x_1 \supset x \; R \; y \supset x_1 \; R \; y$$

- R *is* left-exact *for E (or E left-exact) if and only if:*

$$x \; R \; y \supset x_1 \; R \; y \supset x \; E \; x_1$$

Say R is right-consistent with E when R^{op} is left-consistent with E, and right-exact when R^{op} is left-exact.

Lemma 3.1 *Let $R : \sigma \twoheadrightarrow \tau \twoheadrightarrow o$ and $S : \sigma \twoheadrightarrow \rho \twoheadrightarrow o$ be relations. Let $E_\tau : \tau \twoheadrightarrow \tau \twoheadrightarrow o$ be a per, and let $W_\tau(y) = y \; E_\tau \; y$ and $W_\sigma : \sigma \twoheadrightarrow o$ be well-behavedness predicates.*

1. *The relation R is E_τ right-consistent if and only if:*

$$y \; E_\tau \; y_1 \supset (x \; R \; y \equiv x \; R \; y_1)$$

2. *If R is E_τ right-exact then $x \; R \; y$ implies $y \; E_\tau \; y$.*

3. *Suppose R is E_τ right-exact. Then R is (W_σ, W_τ) total if and only if:*

$$\forall x\colon \sigma |\, W_\sigma .\, \exists y\colon \tau .\, x \; R \; y$$

4. *If R is E_τ right-exact then $x(R \circ S)z = \exists y\colon \tau |\, W_\tau .\, x \; S \; y \wedge y \; R \; z$.*

Proof: 1. $\supset$ **direction:** both directions of the equality follow from consistency, since E_τ is symmetric.

   $\subset$ **direction:** since we know $x\ R\ y_1$ is equivalent to $x\ R\ y$, one certainly follows from the other.

2. This is immediate from exactness.

3. $\supset$ **direction:** this is immediate.

   $\subset$ **direction:** we know $x\ R\ y$, and so by part 2 of the lemma, $W_\tau(y)$ holds.

4. $\supset$ **direction:** we know that $x\ R\ y$, hence by part 2 of the lemma $W_\tau(y)$ holds.

   $\subset$ **direction:** this is immediate.

□

Part 4 of this lemma shows why we did not bother to define "composition up to": it is exactly the same as normal composition in the cases in which we shall use it. The corresponding results hold for R^{op}, of course.

Lemma 3.2 *Let $R : \sigma \twoheadrightarrow \tau \twoheadrightarrow o$ be a binary relation. Let $E_\sigma : \sigma \twoheadrightarrow \sigma \twoheadrightarrow o$ be a per. Let $W_\sigma(x) = x\ E_\sigma\ x$ and $W_\tau : \tau \twoheadrightarrow o$ be well-behavedness predicates.*

Suppose R is (W_σ, W_τ) total, and (E_σ, W_τ) left-consistent and -exact. Then there is the following relationship between R and E_σ:

$$(x\ E_\sigma\ x_1) \iff (\exists y{:}\,\tau.\, x\ R\ y \wedge x_1\ R\ y)$$

If R is (W_σ, W_τ) surjective, and (E_τ, W_σ) right-consistent and -exact then:

$$(y\ E_\tau\ y_1) \iff (\exists x{:}\,\sigma.\, x\ R\ y \wedge x\ R\ y_1)$$

Proof: We prove the first part of the lemma. The second part follows from the first part applied to R^{op}.

The $\subset$ direction follows immediately from the exactness of R. Now for the $\supset$ direction. By totality-up-to, there certainly is a (well-behaved) y related to x. But by consistency, this y must also be related to x_1. □

There is another, more concise, formulation of the notion of total-up-to and left-exact and -consistent.

Definition 3.7 *Let $R : \sigma \twoheadrightarrow \tau \twoheadrightarrow o$ be a binary relation, and let $E_\tau : \tau \twoheadrightarrow \tau \twoheadrightarrow o$ be a per. Let $W_\sigma : \sigma \twoheadrightarrow o$ and $W_\tau : \tau \twoheadrightarrow o$ be well-behavedness predicates with $W_\tau(y) = y\ E_\tau\ y$.*

Say R is a total function up to (W_σ, E_τ) when:

$$\forall x{:}\,\sigma|\,W_\sigma.\,\exists y{:}\,\tau|\,W_\tau.\,\forall y_1{:}\,\tau.\,(x\ R\ y_1) \iff (y\ E_\tau\ y_1)$$

Say $S : \tau \twoheadrightarrow \sigma \twoheadrightarrow o$ is a total function from right to left up to (W_σ, E_τ) when S^{op} is a total function up to (W_σ, E_τ). Say R is a bijection up to (E_σ, E_τ) (or an (E_σ, E_τ) bijection) whenever:

- *$W_\sigma(x) = x\ E_\sigma\ x$;*
- *$W_\tau(y) = y\ E_\tau\ y$;*
- *R is a total function up to (W_σ, E_τ);*
- *R is a total function from right to left up to (W_τ, E_σ).*

Lemma 3.3 *Let E_τ be a per. Let W_σ and W_τ be well-behavedness predicates with $W_\tau(y) = y\ E_\tau\ y$. Let $R : \sigma \twoheadrightarrow \tau \twoheadrightarrow o$ be a relation.*

R is a total function up to (W_σ, E_τ) if and only if the following properties hold:

1. *R is (E_τ, W_σ) right-consistent;*
2. *R is (E_τ, W_σ) right-exact;*
3. *R is (W_σ, W_τ) total.*

Proof: $\supset$ direction.

1. Suppose $W_\sigma(x)$, $y\ E_\tau\ y_1$ and $x\ R\ y$. By the total-functionality-up-to of R, there is a y' such that:

$$\forall y_1'{:}\,\tau|\,W_\tau.\,(x\ R\ y_1') \iff (y'\ E_\tau\ y_1')$$

Hence $y'\ E_\tau\ y$. By transitivity, $y'\ E_\tau\ y_1$. Applying the above once more, $x\ R\ y_1$.

2. Suppose $W_\sigma(x)$, $x\ R\ y$ and $x\ R\ y_1$. By the total-functionality-up-to of R, there is a y' such that:

$$\forall y_1'{:}\,\tau|\,W_\tau.\,(x\ R\ y_1') \iff (y'\ E_\tau\ y_1')$$

Hence both $y'\ E_\tau\ y$ and $y'\ E_\tau\ y_1$. By symmetry and transitivity, $y\ E_\tau\ y_1$.

3. Suppose $W_\sigma(x)$. By the total-functionality-up-to of R, there is a well-behaved y' such that:

$$\forall y'_1\!:\tau|\,W_\tau.\,(x\ R\ y'_1) \iff (y'\ E_\tau\ y'_1)$$

Since $W_\tau(y')$, $y'\ E_\tau\ y'$. We can take y'_1 to be y', so $x\ R\ y'$.

$\subset$ **direction.** Let $x : \sigma$ be a well-behaved value. We must exhibit a well-behaved $y : \tau$ such that:

$$\forall y_1\!:\tau|\,W_\tau.\,(x\ R\ y_1) \iff (y\ E_\tau\ y_1)$$

Since R is total up to (W_σ, W_τ), there is a well-behaved y with $x\ R\ y$. We show that this y has the above property.

Let $y_1 : \sigma$ be a well-behaved value. Suppose that $x\ R\ y_1$. Since R is right-exact, $y\ E_\tau\ y_1$. Suppose instead that $y\ E_\tau\ y_1$. Since R is right-consistent, $x\ R\ y_1$.

□

Corollary 3.4 *Let E_σ and E_τ be pers. Let $W_\sigma(x) = x\ E_\sigma\ x$ and $W_\tau(y) = y\ E_\tau\ y$. Let $R : \sigma \twoheadrightarrow \tau \twoheadrightarrow o$ be a relation. R is a bijection up to (E_σ, E_τ) if and only if the following properties hold.*

1. *R is left-consistent with E_σ up to W_τ;*
2. *R is right-consistent with E_τ up to W_σ;*
3. *R is left-exact with E_σ up to W_τ;*
4. *R is right-exact for E_τ up to W_σ;*
5. *R is total up to (W_σ, W_τ)*
6. *R is surjective up to (W_σ, W_τ)*

Proof: $\supset$ **direction.** Properties 2, 4 and 6 follow from the fact that R is a total-function-up-to. The other properties follow from the fact that R is a total-function-up-to from right to left.

$\subset$ **direction** That R is a total-function-up-to follows from properties 2, 4 and 6. That R is a total-function-up-to from right to left follows from the other properties.

□

Intuitively, if R is an (E_σ, E_τ) bijection, it is a bijection between the equivalence classes of E_σ and those of E_τ. That is, for every E_σ equivalence class $[x]$, there is a (unique) E_τ equivalence class $[y]$ such that if $x \in [x]$ and $y \in [y]$ then $x\ R\ y$; and for every E_τ equivalence class $[y]$, there is a unique E_σ equivalence class with a similar property.

Bijections-up-to are not unique (different bijections-up-to map equivalence classes differently). However, if we know how they map equivalence classes, then we know the whole relation. This is captured by the following lemma.

Lemma 3.5 *Let R and S be (E_τ, E_σ) bijections with $R \subseteq S$. Then $R = S$.*

Proof: Given $x : \sigma$ and $y : \tau$, we must show $x\ R\ y \iff x\ S\ y$. Let $W_\sigma(x) = x\ E_\tau\ x$ and $W_\tau(y) = y\ R\ y$.

- $\supset$ **direction:** this is immediate from the fact that $R \subseteq S$.
- $\subset$ **direction:** assume $x\ S\ y$ and show $x\ R\ y$. By the left-exactness of S, $W_\sigma(x)$. By the totality-up-to of R, there is a well-behaved y_1 with $x\ R\ y_1$.

  Since $R \subseteq S$, $x\ S\ y_1$. Now by the right-exactness of S, $y\ E_\tau\ y_1$. Finally, by the right-consistency of R and the fact that $x\ R\ y_1$, $x\ R\ y$ as required.

□

Lemma 3.6 *Suppose $R : \sigma \twoheadrightarrow \tau \twoheadrightarrow o$ is an (E_σ, E_τ) bijection and $S : \tau \twoheadrightarrow \rho \twoheadrightarrow o$ is an (E_τ, E_ρ) bijection. Then $R \circ S$ is an (E_σ, E_ρ) bijection.*

Proof: Let $W_\sigma(x) = x\ E_\sigma\ x$, $W_\tau(y) = y\ E_\tau\ y$ and $W_\rho(z) = z\ E_\rho\ z$. We prove that $R \circ S$ is a (W_σ, E_ρ) total function, and the result follows by considering R^{op}. In turn, this means that we must prove that $R \circ S$ is E_ρ right-consistent and E_ρ right-exact, and that it is (W_σ, W_ρ) total.

- **Consistency:** assume that $z\ E_\rho\ z_1$ and that $x\ (R \circ S)\ z$. By the definition of composition, this means that there is a $y : \tau$ such that $x\ R\ y$ and $y\ S\ z$. By the right consistency of S, $y\ R\ z_1$. Then, by the definition of composition, $x\ R\ z_1$.
- **Exactness:** assume that $x\ (R \circ S)\ z$ and that $x\ (R \circ S)\ z_1$. From this we can deduce that there is a $y : \tau$ with $x\ R\ y$ and $y\ S\ z$, and that there is a $y_1 : \tau$ with $x\ R\ y_1$ and $y_1\ S\ z_1$.

  By right exactness of R, $y\ E_\tau\ y_1$. By left consistency of S, $y_1\ R\ z$. Finally, by right exactness of S, $z\ E_\rho\ z_1$, which is the result we require.

Totality-up-to: by lemma 3.1 (3), we need only prove:

$$\forall x{:}\, \sigma | W_\sigma .\, \exists z{:}\, \rho .\, x\ (R \circ S)\ z$$

Let $x : \sigma$ be an arbitrary value such that $W_\sigma(x)$.

By the totality-up-to of R, there is a y such that $W_\tau(y)$ and $x\ R\ y$. By the totality-up-to of S, there is a z such that $W_\rho(z)$ and $y\ S\ z$. Thus $x\ (R \circ S)\ z$, and z is the required value.

□

There is a third way to characterize bijections-up-to.

Definition 3.8 *Let $R : \sigma \twoheadrightarrow \tau \twoheadrightarrow o$ be a relation.*

Say R is many-step closed when, for $x, x' : \sigma$ and $y, y' : \tau$, the following holds:

$$x'\ R\ y \supset x'\ R\ y' \supset x\ R\ y' \supset x\ R\ y$$

Define the many-step closure of R to be the smallest many-step closed relation containing R. Define $R^?$ by induction on the following rules:

$$\frac{x\ R\ y}{x\ R^?\ y}$$

$$\frac{x'\ R^?\ y \quad x'\ R^?\ y' \quad x\ R^?\ y'}{x\ R^?\ y}$$

Given a relation R, define a relation $\mathsf{lper}(R) : \sigma \twoheadrightarrow \sigma \twoheadrightarrow o$ as follows:

$$x\ \mathsf{lper}(R)\ x' \iff \exists y{:}\, \tau .\, x\ R^?\ y \wedge x'\ R^?\ y$$

Similarly, define $\mathsf{rper}(R) : \tau \twoheadrightarrow \tau \twoheadrightarrow o$ as $\mathsf{lper}(R^{op})$.

Lemma 3.7 *1. $R^?$ is many-step closed.*

2. $R^?$ is the many-step closure of R.

3. If R is many-step closed, $R^? = R$.

4. $\_^?$ is idempotent. That is, $R^{?^?} = R^?$.

5. If R is many-step closed, then R^{op} is many-step closed.

Proof: 1. The relation $R^?$ is many-step closed by virtue of the second rule defining $R^?$.

2. Let F be some many-step closed relation which includes R. We prove by induction on the definition of $R^?$ that for any x and y with $x\ R^?\ y$, it is the case that $x\ F\ y$.

   First rule Suppose $x\ R\ y$. Since F includes R, $x\ F\ y$.

   Second rule Suppose $x'\ F\ y$, $x'\ F\ y'$ and $x\ F\ y'$. Since F is many-step closed, it must be the case that $x\ F\ y$.

3. The smallest set (of any sort) which includes R is R. It is also many-step closed. It must, therefore, be the smallest many-step closed set to include R.

4. Since $R^?$ is many-step closed, the result follows by part (3) of the lemma.

5. Assume $y'\ R^{op}\ x$, $y'\ R^{op}\ x'$ and $y\ R^{op}\ x'$. Since R is many-step closed, it is the case that $x\ R\ y$, and so $y\ R^{op}\ x$.

□

Lemma 3.8 *Given some relation R, $\mathsf{lper}(R)$ and $\mathsf{rper}(R)$ are pers.*

Proof: We prove that $\mathsf{lper}(R)$ is a per. That $\mathsf{rper}(R)$ is a per follows from consideration of R^{op}.

Symmetry. By definition, $x\ \mathsf{lper}(R)\ x'$ is:

$$\exists y{:}\,\tau.\ x\ R^?\ y \wedge x'\ R^?\ y$$

By the commutativity of $\wedge$, this is also $x'\ \mathsf{lper}(R)\ x$.

Transitivity. We know that $x\ \mathsf{lper}(R)\ x'$ and $x'\ \mathsf{lper}(R)\ x''$:

$$\exists y{:}\,\tau.\ x\ R^?\ y \wedge x'\ R^?\ y$$

$$\exists y'{:}\,\tau.\ x'\ R^?\ y' \wedge x''\ R^?\ y'$$

We show that this y has the following property:

$$x\ R^?\ y \wedge x''\ R^?\ y$$

We know that $x\ R^?\ y$. By the fact that $R^?$ is many-step closed, together with the facts $x''\ R^?\ y'$, $x'\ R^?\ y'$ and $x'\ R^?\ y$, we know that $x''\ R^?\ y$.

□

Lemma 3.9 *Let $R : \sigma \twoheadrightarrow \tau \twoheadrightarrow o$ be a relation. Let $E_\sigma : \sigma \twoheadrightarrow \sigma \twoheadrightarrow o$ and $E_\tau : \tau \twoheadrightarrow \tau \twoheadrightarrow o$ be pers. Let $W_\sigma(x) = x\ E_\sigma\ x$ and $W_\tau(y) = y\ E_\tau\ y$. Let $x\ R^\#\ y = W_\sigma(x) \wedge W_\tau(y) \wedge x\ R\ y$.*

1. *If R is many-step closed and $E_\sigma = \mathsf{lper}(R)$ and $E_\tau = \mathsf{rper}(R)$, then R is a bijection up to (E_σ, E_τ).*
2. *If R is a total function up to (W_σ, E_τ), then $R^\#$ is many-step closed.*
3. *If R is a bijection up to (E_σ, E_τ), then $E_\sigma = \mathsf{lper}(R^\#)$ and $E_\tau = \mathsf{rper}(R^\#)$.*

Proof:

1. We show that R is a total function up to (W_σ, E_τ). That R is a total-function-up-to from right to left follows from the fact that R^{op} is many-step closed.

   For every well-behaved x, we must exhibit a well-behaved y such that the following holds:

   $$\forall y_1 : \tau | W_\tau .\, x\ R\ y_1 \iff y\ E_\tau\ y_1$$

   Since $E_\sigma = \mathsf{lper}(R)$, $W_\sigma(x)$ boils down to $\exists y : \tau.\, x\ R\ y$. We show that this y satisfies the above property.

   First, note that y is well-behaved: since $x\ R\ y$, then $\exists x : \sigma.\, x\ R\ y$. Now, let $y_1 : \tau$ be some well-behaved value. Since $E_\tau = \mathsf{rper}(R)$, we must show:

   $$x\ R\ y_1 \iff \exists x' : \sigma.\, x'\ R\ y \wedge x'\ R\ y_1$$

   $\supset$ **direction.** Assume $x\ R\ y_1$. Then x has the property that $x\ R\ y \wedge x\ R\ y_1$.

   $\subset$ **direction.** Assume $x'\ R\ y$ and $x'\ R\ y_1$. Since $x\ R\ y$, by many-step closedness $x\ R\ y_1$.

2. Assume that R is a total function up to (W_σ, E_τ). Given $x'\ R^\#\ y$, $x'\ R^\#\ y'$ and $x\ R^\#\ y'$, we must show $x\ R^\#\ y$.

   It is immediate from the assumptions that x, x', y and y' are well-behaved. It remains to show that $x\ R\ y$. Since $x'\ R\ y$, $x'\ R\ y'$ and R is right-exact, it is the case that $y\ E_\tau\ y'$. Since $x\ R\ y'$ and R is right-consistent, it is the case that $x\ R\ y$.

3. We show $x\ E_\sigma\ x' = x\ \mathsf{lper}(R^\#)\ x'$. That $E_\tau = \mathsf{rper}(R^\#)$ follows by considering R^{op}.

- $\supset$ **direction.** Assume $x\ E_\sigma\ x'$. Since E_σ is a per, x and x' are well-behaved. Since R is total up to (W_σ, W_τ), there is a well-behaved y with $x\ R\ y$. It remains to show that $x'\ R\ y$. This follows from the left-consistency of R and the fact that $x\ E_\sigma\ x'$.
- $\subset$ **direction.** Assume that $\exists y{:}\,\tau.\, x\ R^{\#}\ y \wedge x'\ R^{\#}\ y$. Since $x\ R^{\#}\ y$, y is well-behaved. The result follows from the left-exactness of R.

$\square$

For relations with type $\tau \twoheadrightarrow \tau \twoheadrightarrow o$, bijectivity-up-to coincides with partial equivalence.

Lemma 3.10 *Suppose we have a relation $E : \tau \twoheadrightarrow \tau \twoheadrightarrow o$. E is a per if and only if it is an (E, E) bijection.*

Proof: The proof is by simple manipulation of the definitions, and is omitted. $\square$

Part I

Logical Frameworks

Chapter 4

Coding binding and substitution explicitly in Isabelle

I am bound
Upon a wheel of fire, that mine own tears
Do scald like molten lead. —William Shakespeare, *King Lear*

Logical frameworks provide powerful methods of encoding object-logical binding and substitution using meta-logical λ-abstraction and application. However, there are some cases in which these methods are not general enough: in such cases object-logical binding and substitution must be explicitly coded. McKinna and Pollack [MP93] give a novel formalization of binding, where they use it principally to prove meta-theorems of Type Theory. We analyze the practical use of McKinna-Pollack binding in Isabelle object-logics, and illustrate its use with a simple example logic.

A version of this chapter has been previously published as a paper of the same name in the 1995 Isabelle Users Workshop [Owe95].

4.1 Introduction

In this chapter we address the problem of coding logics in Isabelle (and other logical frameworks). In some logics, meta-logical binding is unsuitable for coding object-logical binding. For example, as we explain later, logics for programming languages which include features such as pattern matching are difficult to code using meta-binding. We advocate the use of the binding system due to McKinna and Pollack [MP93] to code binding explicitly in these cases. The aim of this chapter is to give enough meta-theoretical background and practical examples to facilitate the coding of a logic in Isabelle using McKinna-Pollack binding, and to motivate and evaluate the choice of McKinna-Pollack binding.

In the remainder of this section, we discuss some motivating examples, and compare systems which allow us to code binding and substitution explicitly, explaining why we choose the system due to McKinna and Pollack. In Section 4.2 we describe McKinna-Pollack binding and give an example of its use, together with several meta-theorems which capture its behaviour. In Section 4.3 we expand on the example, and discuss features of the implementation of the system in Isabelle. In Section 4.4, we return to our motivating concern and examine how McKinna-Pollack binding can encode logics for programming languages. Finally, in Section 4.5, we summarize our results and evaluate the possible uses of McKinna-Pollack binding style in Isabelle.

Most Isabelle code is omitted. However, this should not disguise the fact that this chapter is about an Isabelle implementation. The logic is built on Isabelle HOL. When we give a grammar, it is implemented in Isabelle as the obvious datatype declaration. Similarly, inductive definitions are coded in the obvious way in Isabelle, as are primitive recursive definitions. Where the translation from the page to Isabelle is not immediate, the implementation is explained. The Isabelle code is included in Appendix A: this includes the code for all the proofs in this chapter.

4.1.1 Motivation

Much research has been devoted to coding logics in logical frameworks and generic theorem provers [PE88, DFH95]. Most of these encodings use meta-level λ-abstraction to encode object-level binding, meta-application to encode object-substitution, and meta-variables to represent object variables. However, some logics can be difficult or cumbersome to encode in this manner. In particular, difficulties with logics for programming languages arise because of their sophisticated binding and scoping features.

Throughout the rest of this chapter we will use an encoding of a version of the Simple Theory of Types [Chu40] to provide examples. Judgements are of the form $\Gamma \vdash P$, where Γ is a list of declarations either of the form $v : \tau$ (variable v has type τ) or $v : \sigma = M$ (variable v is term M of (polymorphic) type σ). Declarations may make reference to earlier declarations, for example in the context $x : \tau \to \tau\, ;\ y : \tau \to \tau = \lambda b{:}\, \tau.\, xb$, the declaration of y refers to the earlier declaration of x.

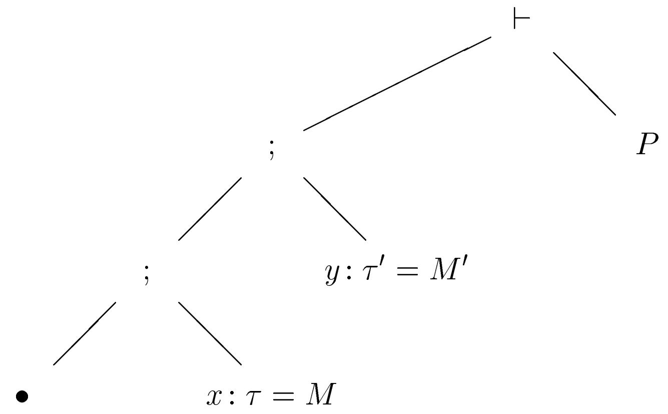
Consider this judgement:

$$x : \tau = M\, ;\ y : \tau' = M' \vdash P$$

If we were to use meta-binding to represent the binding of x and y, we might encode this judgement in the following way:

$$\mathsf{ValDec}(M, \lambda x.\, \mathsf{ValDec}(M', \lambda y.\, \mathsf{Judge}(P)))$$

However, as we shall see, type-checking rules and rules of the logic are only concerned with the last declaration in the context, which in this coding is the inner-most declaration: it is not clear how to express such rules with this coding. We would like to represent the judgement as the obvious abstract syntax tree (the empty context is written $\bullet$):



In order to use this scheme, we must encode the binding that occurs in declarations explicitly. Adding further scoping features—let-expressions, local declarations, let rec declarations—accentuates the problem.

Now we show that we cannot use meta-binding to bind variables *within* terms. Consider a logic for a language which includes pattern matching, having a construct such as the Standard ML `case` expression. Each pattern can bind an arbitrary number of variables, and so we must "compile" a case-expression into some internal form, where each variable is bound separately. Let us take as an example the following expression:

$$\texttt{case}\ e\ \texttt{of}\ \texttt{(}x\texttt{,}\ y\texttt{)}\ \texttt{=>}\ x\ \texttt{+}\ y$$

This might compile into the following form:

$$\mathsf{CaseExp}(e, \mathsf{PatBind}(\lambda x.\, \mathsf{PatBind}(\lambda y.\, \mathsf{PatAndExp}(\texttt{(}x\texttt{,}\ y\texttt{)}\texttt{,}\ x\ \texttt{+}\ y))))$$

This translation would be done by print- and parse-translations. The same ML expression is also represented by another compiled form:

$$\mathsf{CaseExp}(e, \mathsf{PatBind}(\lambda y.\, \mathsf{PatBind}(\lambda x.\, \mathsf{PatAndExp}(\texttt{(}x\texttt{,}\ y\texttt{)}\texttt{,}\ x\ \texttt{+}\ y))))$$

There are two objections to representing pattern matching in this manner:

- Although the two compiled forms above would appear the same after print translation, they cannot be unified. Furthermore, we must add rules which equate the different compiled forms of an expression.

- The parse- and print-translations are difficult to program. More importantly, it is very difficult to establish that they have been programmed correctly.

These problems arise from the fact that pattern matching should be a *single* binding which binds many variables at once. This is not the same as multiple bindings of single variables. With such a "compiled" coding, it is difficult to satisfy ourselves that the logic actually represents expressions as we intend.

By coding binding explicitly, we can represent `case`-expressions just as the grammar of the language suggests:

$$\mathsf{CaseExp}(e, \mathsf{Match}(\texttt{(}x\texttt{, }y\texttt{), }x\texttt{ + }y))$$

The translations are trivial, each syntactically distinct expression has a unique representation, and the correctness of the representation is a non-issue.

4.1.2 Systems of explicit binding and substitution

Rather than using increasingly arcane and counter-intuitive codings, our approach is to axiomatize binding. We consider three possible systems.

De Bruijn binding [dB72] eliminates α-conversion by using indices rather than names for bound variables—terms which are α-variants in name-carrying systems are syntactically identical in de Bruijn systems. De Bruijn binding is widely used, for example in Type Theory (for example by Altenkirch [Alt93]), and in theorem provers (including Isabelle [Pau94a]).

Unfortunately, de Bruijn terms are very difficult to read, and they are usually translated to and from a name-carrying form for display and input. This can be done by providing a software front-end to perform the translation, but Gordon [Gor93] encodes the translation within the framework of the theorem prover. However, any method of translation adds another layer of processing on top of the naked de Bruijn terms: this adds complexity and reduces clarity and abstraction. It is also the case that many formal systems are defined using names (for example, Standard ML [MTH90]), and we should be wary of the faithfulness of an encoding of a logic based on such a formal system if the encoding does not use names.

The "standard" method of binding is originally due to Curry and Feys [CF58] and is given a clear exposition by Hindley and Seldin [HS86]. Standard binding systems define the substitution $(\lambda x.\, M)[N/y]$ as $\lambda z.\, M[z/x][N/y]$ for some fresh z (at least in the case that capture would occur). Stoughton [Sto88] observes that since $M[z/x]$ is not a sub-term of $\lambda x.\, M$, substitution cannot be defined by recursion on the structure of terms, and must instead be defined by recursion on the size of terms. This complicates proof considerably. He proposes a system which performs the substitutions in parallel, $M[z/x, N/y]$, which means that substitution *can* then be defined by recursion on the structure of terms, since M certainly is a sub-term of $\lambda x.\, M$.

We regard this difficulty in defining substitution as a clue that the substitution operation is the wrong place to worry about variable capture, and that capture should instead be avoided in the construction of the terms, allowing substitution to be defined simply. The system used in this chapter is due to McKinna and Pollack [MP93]: it uses names for bound variables. A benefit of taking care over term construction is that α-conversion is rarely necessary: it comes "for free".

4.2 McKinna-Pollack binding

McKinna and Pollack give a system of binding and substitution in which variables are divided into two disjoint classes, intended to represent free and bound variables. The two classes of variable are referred to as *f-variables* (f, f', intended to represent free variables—McKinna and Pollack call these parameters) and *b-variables* (b, b', representing bound variables—called simply variables by McKinna and Pollack). It is syntactically impossible for an f-variable to be bound in a term, and it is an important meta-theorem (to be formalized later as Theorem 4.7) that no b-variable occurs free in a valid deduction.

In McKinna-Pollack binding, both classes of variables are names and not de Bruijn indices. Specifically, all we know about them is that there are infinitely many of them (in the sense that one can always pick a fresh one). Typically, f- and b-variables will be picked from the same set of identifiers, with a constructor wrapped around them. Isabelle itself splits variables into free and bound variables [Pau94b, Section 6.5], but in Isabelle, bound variables are represented by de Bruijn indices.

The grammar for simply-typed lambda terms which we will use in our example logic is given in Figure 4.1. We assume that the set of constants is disjoint from the sets of b- and f-variables. We write object equality as "$\approx$" in order to

$$
\begin{array}{lcl}
ty & ::= & \Omega \\
 & & ty_1 \to ty_2 \\
 & & \\
constant & ::= & \supset \\
 & & \varepsilon \\
 & & \approx \\
 & & \\
term & ::= & fvar \\
 & & bvar \\
 & & constant \\
 & & \lambda bvar\colon ty.\, term \\
 & & term_1\; term_2
\end{array}
$$

Figure 4.1: simply-typed lambda terms.

$$
\begin{array}{lll}
(BSubstF) & f[N/b] & = f \\
(BSubstB) & b'[N/b] & = \text{if}\,(b = b',\ N,\ b') \\
(BSubstC) & const[N/b] & = const \\
(BSubstLda) & (\lambda b'\colon \tau.\, M)[N/b] & = \text{if}\,(b = b',\ \lambda b'\colon \tau.\, M,\ \lambda b'\colon \tau.\, M[N/b]) \\
(BSubstApp) & (M\, M')[N/b] & = M[N/b]\, M'[N/b]
\end{array}
$$

Figure 4.2: b-substitution.

differentiate it from Isabelle meta-equality ($\equiv$) and Isabelle HOL equality ($=$). Hilbert's description operator ("any") is written ε. The type Ω is the type of truth-values. Terms with no free b-variables are called *b-closed*.

4.2.1 Substitution and closedness

There are two notions of substitution. Substitution of a term for a b-variable (b-substitution, written $M[N/b]$) is defined by primitive recursion on terms by the rules in Figure 4.2. Since f-variables, b-variables and constants are disjoint, b-substitution at f-variables and constants does nothing. As in standard substitution, a binding of a variable will shadow substitution for it (the rule *BSubstLda* with $b = b'$). However, unlike standard substitution, we do not α-convert to avoid capture in the substituted term (*BSubstLda* with $b \neq b'$). This is because we know that a b-closed term has no free b-variables—and, in particular, the variable bound by the λ-abstraction is not free in the substituted term.

Substitution of a term for an f-variable (f-substitution, written $M[f = N]$) is defined by primitive recursion on terms by the rules in Figure 4.3. This is simply

$$
\begin{array}{llcl}
(\mathit{FSubstF}) & f'[f=N] & = & \mathrm{if}\,(f=f',\,N,\,f') \\
(\mathit{FSubstB}) & b[f=N] & = & b \\
(\mathit{FSubstC}) & \mathit{const}[f=N] & = & \mathit{const} \\
(\mathit{FSubstLda}) & (\lambda b{:}\,\tau.\,M)[f=N] & = & \lambda b{:}\,\tau.\,M[f=N] \\
(\mathit{FSubstApp}) & (M\,M')[f=N] & = & M[f=N]\,M'[f=N]
\end{array}
$$

Figure 4.3: f-substitution.

$$
\begin{array}{ll}
(\mathit{BClosedF}) & \dfrac{}{f \in \mathsf{BClosed}} \\[2ex]
(\mathit{BClosedC}) & \dfrac{}{\mathit{const} \in \mathsf{BClosed}} \\[2ex]
(\mathit{BClosedLda}) & \dfrac{M[f/b] \in \mathsf{BClosed} \quad f \notin M}{\lambda b{:}\,\tau.\,M \in \mathsf{BClosed}} \\[2ex]
(\mathit{BClosedApp}) & \dfrac{M \in \mathsf{BClosed} \quad M' \in \mathsf{BClosed}}{M\,M' \in \mathsf{BClosed}}
\end{array}
$$

Figure 4.4: the set BClosed.

textual substitution: everywhere the f-variable f appears in M, it is replaced by N. In particular, substitution at λ-abstractions does not need to take into account shadowing of the substitution by the binding: since λ can only bind b-variables, it can never shadow a substitution for an f-variable. We can think of f-variables as "holes" in terms, and f-substitution $M[f = N]$ as plugging a term N into all the holes labelled f in M.

The following theorem shows the connection between the two kinds of substitution. We write $b \in M$ to mean that b-variable b occurs free in M, and $f \in M$ to mean that f-variable f occurs in M—effectively meaning "occurs *free*" because f-variables can never be bound in terms. The formal definitions are left to the interested reader: they are very simple.

Theorem 4.1 (Factorisation.) *Given $f \notin M$:*

$$M[N/b] = M[f/b][f = N]$$

Proof: By induction on M. □

We are now in a position to give a formal definition of b-closed terms. The set BClosed is given inductively by the rules in Figure 4.4.

In the rule *BClosedLda* we require $f \notin M$: this is the usual form for rules in the McKinna-Pollack style. We are careful to maintain b-closedness of terms (free occurrences of b have been substituted away in $M[f/b]$). The side-condition ensures that b is not identified with any f-variable occurring in M by the substitution $M[f/b]$—this would be a form of variable capture.

In this case we are only concerned with the b-closedness of the term, and so do not care if f and b are identified. We can define $\mathsf{BClosed}'$ in exactly the same way as $\mathsf{BClosed}$, but omitting the side-condition on the λ rule (McKinna and Pollack use this definition of b-closedness). The sets $\mathsf{BClosed}$ and $\mathsf{BClosed}'$ can then be proved identical by a messy induction on the size of terms. In general, however, when we define a relation we wish to avoid this kind of capture, and so the side-condition cannot be omitted.

Theorem 4.2 (Substitution preserves b-closedness.)

1. *For any f-variable f, if M and N are b-closed terms, then $M[f = N]$ is b-closed.*

2. *If $\lambda b{:}\,\tau.\,M$ and N are b-closed terms, then $M[N/b]$ is b-closed.*

The converse of neither part of this theorem holds: if $f \notin M$, then $M[f = N] = M$, which tells us nothing about N, and similarly if $b \notin M$, $M[N/b] = M$.

An attempt to prove this theorem by induction on the definition of $\mathsf{BClosed}$ will fail. This is because in the rule *BClosedLda*, we only require that there *exists* a suitable f. The induction hypothesis for the λ case tells us that there is an f' such that $M[f'/b][f = N] \in \mathsf{BClosed}$, but we are attempting to prove $M[f = N][f'/b] \in \mathsf{BClosed}$ for *every* f'. This problem is common when proving properties of McKinna-Pollack-style relations.

Intuitively, since $M[f/b]$ is b-closed for some fresh f, it is in fact b-closed for every possible f. We define the set $\mathsf{BClosed}''$ in the same way as $\mathsf{BClosed}$, but with the following λ rule:

$$(\mathit{BClosedBLda})\quad \frac{\forall f. M[f/b] \in \mathsf{BClosed}''}{\lambda b{:}\,\tau.\,M \in \mathsf{BClosed}''}$$

Surprisingly, it is not necessary to require f to be fresh. The quantifier in the hypothesis of this rule means that $\mathsf{BClosed}''$ cannot (currently) be declared as an Isabelle inductive set: it must be hand-coded as a least fixed point. We now show that the sets $\mathsf{BClosed}''$ and $\mathsf{BClosed}$ are identical. It is immediate that $\mathsf{BClosed}'' \subseteq \mathsf{BClosed}$, since if the hypothesis holds for every f, there certainly exists a fresh f such that it holds. To prove $\mathsf{BClosed} \subseteq \mathsf{BClosed}''$, we must introduce the notion of *renaming*. A renaming is a finite map from f-variables to f-variables. We lift renamings ρ to be operations on terms, $\rho^*(\_)$, in the obvious way.

Theorem 4.3 *If $M \in \mathsf{BClosed}$, then, for all renamings ρ, $\rho^*(M) \in \mathsf{BClosed}''$.*

Proof: By induction on the definition of $\mathsf{BClosed}$. All the cases are simple, except the λ case. This case boils down to deducing $\rho_0^*(M)[f_0/b] \in \mathsf{BClosed}''$ for arbitrary ρ_0, f_0. The induction hypothesis is $\forall \rho.\, \rho^*(M[f'/b]) \in \mathsf{BClosed}''$, where $f' \notin M$. We instantiate ρ in the induction hypothesis with $\rho_0 + \{f' \mapsto f_0\}$, and the result follows by equational reasoning. □

Corollary 4.4

$$\mathsf{BClosed} = \mathsf{BClosed}''$$

Proof: We already know $\mathsf{BClosed}'' \subseteq \mathsf{BClosed}$. We can deduce $\mathsf{BClosed} \subseteq \mathsf{BClosed}''$ from the previous theorem with $\rho = \emptyset$. □

Such proofs are generally possible for relations defined in McKinna-Pollack style. We can now use the induction rule for $\mathsf{BClosed}''$ to prove properties of $\mathsf{BClosed}$, as required.

Finally, we are in a position to prove Theorem 4.2.

Proof of Theorem 4.2: Part 1 is an induction on $\mathsf{BClosed}''$. It requires the following lemma:

$$N' \in \mathsf{BClosed} \supset M[N/b][f = N'] = M[f = N'][N[f = N']/b]$$

In turn, this requires the lemma $N \in \mathsf{BClosed} \supset M[N/b] = M$.

Part 2 follows from part 1 by the Factorisation Theorem (Theorem 4.1). □

We show that $\mathsf{BClosed}$ correctly formalizes the set of b-closed terms.

Theorem 4.5 *Given a term M, $M \in \mathsf{BClosed}$ if and only if $b \notin M$ for every b-variable b.*

Proof: This is proved by induction on the size of M. □

We have no further use for the test $b \in M$, and only use $\mathsf{BClosed}$ from now on. Before moving on, note that we could have defined $b \in M$ to be $f \in M[f/b]$, where $f \notin M$.

4.2.2 Type-checking and contexts

Having developed the machinery of McKinna-Pollack substitution, we move on to a first application: type-checking.

The type system is an adaptation of HOL's. The grammar for types allows only monomorphic types: generalized types such as $A \to A$ (the type of functions from any type to itself) are admitted by using Isabelle free meta-variables to stand for types, (so A is an Isabelle free meta-variable in $A \to A$). The constants are genuinely polymorphic; that is, they may have more than one type.

Declarations of terms are also polymorphic. They are written:

$$f : \forall \alpha_1, \ldots, \alpha_n.\, \tau = M$$

This indicates that M may have any type of the form τ' where τ' is τ with each α_i replaced by a type. The variables α_i may also appear in M, as in the declaration $f : \forall \alpha.\, \alpha \to \alpha = \lambda b{:}\, \alpha.\, b$. In fact, then, $f : \forall \alpha_1, \ldots, \alpha_n.\, \tau = M$ is simply syntactic sugar for $\mathsf{Val}(f, S)$ where S is the *declaration scheme* $\forall \alpha_1, \ldots, \alpha_n.\, \langle\langle \tau, M \rangle\rangle$. Binding of the α_i in declaration schemes is coded as Isabelle meta-binding in the usual Isabelle way. Declaration schemes are formed from two constructors: basic schemes are just types paired with a term of that type; abstracted schemes are schemes quantified over a variable:

```
BasicSch    :: "[Ty, Term] => Scheme"       ("<<_, _>>")
AbsSch      :: "(Ty => Scheme) => Scheme"   (binder "SCH " 100)
```

The relation $\mathsf{InstType}(\forall \alpha_1, \ldots, \alpha_n.\, \langle\langle \tau, M \rangle\rangle, \tau')$ holds when τ' can be obtained from τ by instantiating the variables $\alpha_1, \ldots, \alpha_n$.

Similarly, $\mathsf{InstTerm}(\forall \alpha_1, \ldots, \alpha_n.\, \langle\langle \tau, M \rangle\rangle, M')$ holds when M' can be obtained from M by instantiating $\alpha_1, \ldots, \alpha_n$. Their definitions are omitted, but are simple.

Declarations of types for f-variables are written $f : \tau$: the type τ is not polymorphic. Contexts are lists of declarations of either form. Notice that contexts declare terms and types for f-variables, not b-variables.

The typing judgement is defined inductively by the rules in Figure 4.5.

Now we define what it means for a context Γ to be valid, written $\vdash \Gamma$. We say a declaration $f : \forall \alpha_1 \ldots \alpha_n.\, \tau = M$ is well-typed in Γ if and only if for every instance τ' of τ and corresponding instance M' of M, $\Gamma \vdash M' : \tau'$. Declarations of the form $f : \tau$ are always considered to be well-typed. A context $\Gamma;\ dec$ is valid exactly when Γ is valid and dec is well-typed in Γ (the empty context is valid, too, of course).

$$
\begin{array}{lcl}
\mathsf{TypesOf}(\supset) & = & \{\Omega \to \Omega \to \Omega\} \\
\mathsf{TypesOf}(\varepsilon) & = & \{(A \to \Omega) \to A, A \text{ is a type}\} \\
\mathsf{TypesOf}(\approx) & = & \{A \to A \to \Omega, A \text{ is a type}\}
\end{array}
$$

$$(\mathit{LookupVbl}) \qquad \frac{}{\tau \in \mathsf{LookupTypes}(\Gamma;\ f : \tau, f)}$$

$$(\mathit{LookupVal}) \qquad \frac{\mathsf{InstType}(S, \tau)}{\tau \in \mathsf{LookupTypes}(\Gamma;\ \mathsf{Val}(f, S), f)}$$

$$(\mathit{LookupWeak}) \qquad \frac{\tau \in \mathsf{LookupTypes}(\Gamma, f)}{\tau \in \mathsf{LookupTypes}(\Gamma;\ \mathit{dec}, f)} \qquad f \text{ not declared by } \mathit{dec}$$

$$(\mathit{TypF}) \qquad \frac{\tau \in \mathsf{LookupTypes}(\Gamma, f)}{\Gamma \vdash f : \tau}$$

$$(\mathit{TypC}) \qquad \frac{\tau \in \mathsf{TypesOf}(\mathit{const})}{\Gamma \vdash \mathit{const} \colon \tau}$$

$$(\mathit{TypLda}) \qquad \frac{\Gamma;\ f : \tau \vdash M[f/b] : \tau' \quad f \notin M}{\Gamma \vdash \lambda b \colon \tau.\, M : \tau \to \tau'}$$

$$(\mathit{TypApp}) \qquad \frac{\Gamma \vdash M : \tau \to \tau' \quad \Gamma \vdash M' : \tau}{\Gamma \vdash M\ M' : \tau'}$$

Figure 4.5: the typing judgement.

The typing rules only deduce a type for b-closed terms. This is formalized as the following theorem.

Theorem 4.6

1. *If* $\Gamma \vdash M : \tau$ *then* $M \in \mathsf{BClosed}$.
2. *If* $\vdash \Gamma$ *then, for every declaration of the form* $f : \forall \alpha_1 \ldots \alpha_n . \tau = M$ *occurring in* Γ, $M \in \mathsf{BClosed}$.

Proof:

1. By induction on the definition of the typing judgement.
2. A simple corollary of part 1, since $\vdash \Gamma$ means that all terms in Γ are well-typed.

$\square$

4.3 A version of the Simple Theory of Types

The version of the Simple Theory of Types we implement is similar to that implemented in HOL [GM93] and to an early version of Isabelle HOL [Pau90]. It contains both λ-binding and also binding of terms in declarations within a context, and so illustrates the use of McKinna-Pollack binding.

Judgements in the logic are of the form $\Gamma \vdash P$, meaning "P holds in the presence of context Γ." Recall that contexts give types to f-variables and declarations of values for f-variables. We ensure type-correctness of judgements by ensuring that axioms are well typed, and that rules preserve well-typedness, as is usual.

The rules and axioms of the system are given in Figure 4.6. Rules whose only hypotheses are well-formedness conditions are considered to be axioms.

4.3.1 Declarations

We can look-up declarations via the rule *Lookup*:

$$\frac{\Gamma \vdash M \approx M' \quad f \notin M' \quad \mathsf{InstTerm}(S, M)}{\Gamma;\ \mathsf{Val}(f, S) \vdash f \approx M'}$$

Notice that M is evaluated in the context Γ: any occurrences of f in M refer to a previous declaration in Γ.

Axioms

(*EqRefl*) $$\frac{\vdash \Gamma \quad \Gamma \vdash M : \tau}{\Gamma \vdash M \approx M}$$

(*BetaConv*) $$\frac{\vdash \Gamma \quad \Gamma \vdash (\lambda b{:}\, \tau.\, M)\, N : \tau}{\Gamma \vdash ((\lambda b{:}\, \tau.\, M)\, N) \approx M[N/b]}$$

(*OmegaCases*) $$\frac{\vdash \Gamma}{\Gamma \vdash \forall P{:}\, \Omega.\, (P \approx \mathsf{T}) \vee (P \approx \mathsf{F})}$$

(*ImpAntiSym*) $$\frac{\vdash \Gamma}{\Gamma \vdash \forall P, Q{:}\, \Omega.\, (P \supset Q) \supset (Q \supset P) \supset (P \approx Q)}$$

(*EtaConv*) $$\frac{\vdash \Gamma}{\Gamma \vdash \forall H{:}\, \tau \to \tau'.\, (\lambda b{:}\, \tau.\, H\, b) \approx H}$$

(*Select*) $$\frac{\vdash \Gamma}{\Gamma \vdash \forall P{:}\, \alpha \to \Omega.\, \forall x{:}\, \alpha.\, P\, x \supset P\, (\varepsilon\, P)}$$

Rules

(*Subst*) $$\frac{\Gamma \vdash P[f = M] \quad \Gamma \vdash M \approx M'}{P[f = M']}$$

(*Abs*) $$\frac{\Gamma;\ f : \tau \vdash M[f/b] \approx M[f/b'] \quad f \notin M, M'}{\Gamma \vdash \lambda b{:}\, \tau.\, M \approx \lambda b'{:}\, \tau.\, M'}$$

(*ImpI*) $$\frac{\begin{array}{c}[\Gamma \vdash P] \\ \vdots \\ \Gamma \vdash Q\end{array}}{\Gamma \vdash P \supset Q}$$

(*MP*) $$\frac{\Gamma \vdash P \quad \Gamma \vdash P \supset Q}{\Gamma \vdash Q}$$

(*Lookup*) $$\frac{\Gamma \vdash M \approx M' \quad f \notin M' \quad \mathsf{InstTerm}(S, M)}{\Gamma;\ \mathsf{Val}(f, S) \vdash f \approx M'}$$

(*Weak*) $$\frac{\Gamma \vdash P \quad \textit{dec} \text{ well-typed in } \Gamma \quad f \notin P}{\Gamma;\ \textit{dec} \vdash P} \qquad \textit{dec} \text{ declares } f$$

Figure 4.6: The Simple Theory of Types

We give an atomic weakening rule for contexts. The declaration *dec* is a single value or type declaration which declares the f-variable f.

$$\frac{\Gamma \vdash P \quad \mathit{dec} \text{ well-typed in } \Gamma \quad f \notin P}{\Gamma;\ \mathit{dec} \vdash P}$$

This rule is provably equivalent to weakening by many declarations at once.

The logical connectives T, F, $\wedge$, $\vee$, $\neg$, $\forall$ and $\exists$ are declared in a standard context which gives their usual HOL definitions. From these definitions, it is possible to derive the usual natural deduction rules for these connectives.

4.3.2 Equality and conversion

We encode β-reduction in the usual HOL way: we say that two terms are equal if one is obtained from the other by reducing a single outer-most β redex. The other rules for equality mean that it includes the reflexive transitive closure of β-reduction within terms.

McKinna-Pollack binding style does not require us to explicitly define α-conversion. As a side-effect, the rule *Abs* captures α-conversion. For example, $\Gamma \vdash \lambda x{:}\,\tau.\, x \approx \lambda y{:}\,\tau.\, y$, by *EqRefl* followed by *Abs*. (Of course, α-conversion is admitted anyway in the Simple Theory of Types by extensionality, but even in a system where this were not so, α-conversion would still be admitted by *Abs*).

The following is a simple consequence of the axiom *EtaConv*:

$$\frac{\vdash \Gamma \quad \Gamma \vdash H : \tau \to \tau'}{\Gamma \vdash (\lambda b{:}\,\tau.\, H\, b) \approx H}$$

In a standard binding system this rule would have the additional side-condition $b \notin H$—but since H is well-typed, it is b-closed, and so we need not check this.

4.3.3 Deductions

Unfortunately, the judgement $\Gamma \vdash P$ cannot be defined inductively, because of its negative occurrence in the discharged hypothesis in *ImpI*. The only practical problem this causes us is the inability to prove meta-theorems by induction on the definition of $\Gamma \vdash P$ within Isabelle. Informally, we can still do induction on *derivations*—and we could formalize this, as Paulson has done for his proof of the completeness of propositional logic in the Isabelle examples library, but the system would not be usable for real proof.

Theorem 4.7 *(Well-formedness of deductions) Suppose we deduce the following:*

$$\frac{\Gamma_1 \vdash P_1 \quad \ldots \quad \Gamma_n \vdash P_n}{\Gamma \vdash P}$$

1. *If* $\Gamma_i \vdash P_i\colon\Omega$ *for every* i*, then* $\Gamma \vdash P\colon\Omega$.

2. *If* $\vdash \Gamma_i$ *for every* i*, then* $\vdash \Gamma$.

3. *If* $P_i \in \mathsf{BClosed}$ *for every* i*, then* $P \in \mathsf{BClosed}$.

4. *If* $M \in \mathsf{BClosed}$ *for every term* M *in a* Γ_i*, then for every* M *appearing in* Γ*,* $M \in \mathsf{BClosed}$.

Proof: By inspection, each of these properties holds for each of the rules and axioms in Figure 4.6. Hence, by induction on derivations, they hold for all derivations. □

Corollary 4.8 1. *Suppose we deduce the following:*

$$\frac{\Gamma_1 \vdash P_1 \quad \ldots \quad \Gamma_n \vdash P_n}{\Gamma \vdash P}$$

If $\Gamma_i \vdash P_i\colon\Omega$ *for every* i*, then* $\Gamma \vdash P\colon\Omega$*,* $\vdash \Gamma$*,* $P \in \mathsf{BClosed}$*, and* $M \in \mathsf{BClosed}$ *for every* M *occurring in* Γ.

2. *Property 1 and the properties in Theorem 4.7 each hold for every node* $\Gamma \vdash P$ *in a derivation tree.*

Proof: 1. Since $\Gamma_i \vdash P_i\colon\Omega$, then $\vdash \Gamma_i$, $P_i \in \mathsf{BClosed}$ and $M \in \mathsf{BClosed}$ for every M occurring in a Γ_i, and therefore $\Gamma \vdash P\colon\Omega$, $\vdash \Gamma$, $P \in \mathsf{BClosed}$, and $M \in \mathsf{BClosed}$ for every M occurring in Γ.

2. To see that these properties hold of interior nodes of a derivation tree, observe that each interior node is itself the conclusion of a sub-derivation of the tree.

□

4.3.4 Implementation

The Isabelle implementation raises issues relating to giving proofs *using* the logic, rather than simply meta-proofs *about* the logic.

The implementation of the substitutivity rule illustrates some of these issues. The usual McKinna-Pollack coding of substitutivity is given in Figure 4.6. The term P has some "holes" in it (the f-variable f); these holes are filled by M in the antecedent and M' in the conclusion. Since we are using Isabelle, however, we can use meta-substitution in this case: a term P with some holes filled by

M is $P(M)$ in Isabelle. Since the logic does not use meta-binding, this is simply textual substitution, as required. The rule becomes:

$$\frac{\Gamma \vdash M \approx M' \quad \Gamma \vdash P(M)}{\Gamma \vdash P(M')}$$

There is also a problem with the applicability of some rules in their current form. Consider attempting to deduce the following.

$$\Gamma \vdash ((\lambda b\colon \tau.\, b) f) \approx f$$

This is simply a β-reduction. However, we cannot apply the *BetaConv* axiom, since f is not in the form $b[f/b]$. Instead, we must first derive the following rule, which is applicable.

$$\frac{M[N/b] = M' \quad \vdash \Gamma \quad \Gamma \vdash (\lambda b\colon \tau.\, M)\, N : \tau}{\Gamma \vdash ((\lambda b\colon \tau.\, M)\, N) \approx M'}$$

When this rule is applied, we obtain a subgoal $b[f/b] = f$, which can be solved by the simplifier.

Now consider performing the substitution $b'[N/b]$. In the case that b and b' are identical, this can be solved immediately to yield N. Suppose b and b' are not identical (and are intended to be distinct). Since b and b' are both implicitly universally quantified meta-variables, $b \neq b'$ cannot in general be deduced, and so we cannot proceed any further and must leave the substitution in the form $\mathrm{if}\,(b = b',\, N,\, b')$. This is a consequence of the use of meta-variables to represent object-variables. We must add $b \neq b'$ as an assumption.

In fact, the neatest way to do this is to define a predicate $\mathsf{distinct}$, which says that the elements of a list are pair-wise distinct. We now simply add the assumption $\mathsf{distinct}(bs)$ to all proofs, where bs is a list containing all those b-variables which appear in the proof. We must similarly provide the assumption $\mathsf{distinct}(fs)$.

In the theory files accompanying this chapter (see Appendix A), f-variables are formed either from identifiers or from one of the connectives T, F, $\wedge$, $\vee$, $\neg$, $\forall$ and $\exists$. This means that the connectives can be proved to be distinct from each other and all other f-variables, without needing to assert $\mathsf{distinct}(\mathsf{T}, \mathsf{F}, \wedge, \vee, \neg, \forall, \exists, \ldots)$.

Not all of Isabelle's automatic proof procedures may be used with this logic.

4.4 Programming languages

In the introduction to this chapter, we showed that logics for programming languages are difficult to code using meta-binding. We now give examples of how they can be implemented using McKinna-Pollack binding. The features for which we give an account here (let expressions, local declarations, and pattern matching) can be found in many programming languages. We consider a simple eager statically-bound language with ML-like syntax.

The logic we outline is based on the logic of Section 4.3. Logical terms may now include programming language expressions. We also allow the context to contain programming language declarations. This means that a judgement in the logic would typically be a statement about programming language expressions in the presence of programming language declarations.

4.4.1 Let-expressions and local declarations

Let-expressions and local declarations allow us to limit the scope of certain declarations. Let-expressions are of the form:

$$\texttt{let}\ \mathit{decs}\ \texttt{in}\ e\ \texttt{end}$$

Here, *decs* is a list of declarations. It differs from the list of declarations in the context in two ways:

- It declares values for b-variables, not f-variables. McKinna-Pollack binding relies on the fact that no f-variable binding occurs within a b-variable binding, and a let-expression might occur within another construct that binds b-variables (a λ-abstraction, for instance).

- We will primarily be concerned with the left-most declaration in *decs*. So for convenience, we insist that the list associates to the right: the left-most declaration is the outer-most.

The rules for let-expressions allow declarations to be moved to and from the context. They are complicated by the fact that local declarations will mean that a single declaration can bind more than one variable.

One such complication is that we will need a multiple b-substitution operation, $X[f_1 \ldots f_n/b_1 \ldots b_n]$. We only use this operation to substitute f-variables for b-variables, and so we can perform the substitutions of f_i for b_i in any order (or in parallel) since they cannot affect each other.

The operation $dec\langle f_1 \ldots f_n/b_1 \ldots b_n\rangle$ replaces binding occurrences of b_i by f_i, and replaces bound occurrences of b_i with f_i. For example:

$$\begin{gathered}(\texttt{val}\ b_j\ \texttt{=}\ e\texttt{;}\ \mathit{decs})\langle f_1 \ldots f_n/b_1 \ldots b_n\rangle \\ = \\ \texttt{val}\ f_j\ \texttt{=}\ e\texttt{;}\ \mathit{decs}[f_j/b_j]\langle f_1 \ldots f_n/b_1 \ldots b_n\rangle\end{gathered}$$

In order to avoid variable capture, we require that $f_i \neq f_j$ when $b_i \neq b_j$. Thus $dec\langle f_i/b_i\rangle$ is a declaration which declares f-variables $f_1, \ldots, f_n$ and which has exactly the same shape as *dec*.

For brevity, write $X\langle f_1 \ldots f_n/b_1 \ldots b_n\rangle$ as $X\langle f_i/b_i\rangle$ and $X[f_1 \ldots f_n/b_1 \ldots b_n]$ as $X[f_i/b_i]$. Here is one rule for let-expressions, when *dec* declares b-variables $b_1, \ldots, b_n$:

$$\frac{\Gamma;\ dec\langle f_i/b_i\rangle \vdash (\texttt{let}\ decs\ \texttt{in}\ e\ \texttt{end})[f_i/b_i] \approx M \quad f_i \notin decs, M, e}{\Gamma \vdash \texttt{let}\ dec;\ decs\ \texttt{in}\ e\ \texttt{end} \approx M}$$

The rule accounts for the scope of the declaration *dec* by requiring that $f_i \notin M$: this means that *dec* cannot capture variables in M when it is moved to the context.

Local declarations are of the form:

$$\texttt{local}\ decs\ \texttt{in}\ decs'\ \texttt{end}$$

Again, *decs* declares b-variables, and associates to the right. Local declarations can appear in two places:

- They may form part of the context. In this case *decs'* would declare f-variables and would associate to the left (as is usual for declarations in the context).

- Alternatively, they may form part of a let-expression or appear in the local part of a local declaration. In this case, *decs'* would declare b-variables and would associate to the right.

The rules for local declarations move the declarations to and from the context, and are similar to the rules for let-expressions.

We can no longer say that the terms in a declaration are b-closed, as we did in Theorem 4.7 (4). Now, instead, we must say that the declaration as a whole is b-closed. For example `local` *decs* `in` *decs'* `end` is b-closed if *decs* is b-closed and $decs'[f_1 \ldots f_n/b_1 \ldots b_n]$ is b-closed, where *decs* declares $b_1, \ldots, b_n$.

4.4.2 Pattern matching

We consider the case-expression `case` e `of` *match*, where *match* is:

$$pat_1\texttt{=>}e_1 \mid \ldots \mid pat_n\texttt{=>}e_n$$

For simplicity we assume that patterns are exhaustive and non-overlapping.

Each pattern pat_i can bind an arbitrary number of b-variables, which may appear in e_i. Suppose pat_i binds variables $b_{i,1} : \tau_{i,1} \ldots b_{i,m_i} : \tau_{i,m_i}$. Considering pat_i as an expression, we know the following:

$$\forall b_{i,1} : \tau_{i,1} \ldots b_{i,m_i} : \tau_{i,m_i}. \, \texttt{case} \; pat_i \; \texttt{of} \; match \approx e_i$$

Call this term $\mathsf{CaseRule}(i, \texttt{case} \; e \; \texttt{of} \; match)$. It is b-closed by virtue of the fact that the universal quantifier binds every free b-variable which occurs in pat_i. We can thus give the following rule within Isabelle:

$$\frac{\vdash \Gamma \quad P = \mathsf{CaseRule}(i, \texttt{case} \; e \; \texttt{of} \; match) \quad \Gamma \vdash P : \Omega}{\Gamma \vdash P}$$

The term $\mathsf{CaseRule}(i, \texttt{case} \; e \; \texttt{of} \; match)$ is expressed as a primitive recursive function within Isabelle. It is easy to program: the least trivial step it must make is to determine the b-variables bound by pattern i.

4.5 Conclusions

Logics with explicit coding of binding and substitution are considerably more complex than logics which rely on meta-binding. However, they may be necessary if:

- there is no known way to code the logic using meta-binding;
- the encoding using meta-binding is unacceptable (because its correctness is difficult to establish, for example); or
- some proofs will need induction over terms.

These observations hold for all systems with explicit binding, not just McKinna-Pollack binding.

McKinna-Pollack binding has a well-developed and attractive meta-theory. Although rules in this style may look odd at first sight, they are in fact very easy to formulate—with experience they are simpler to formulate than rules for "standard" binding (although terms containing two classes of variables can on occasion be unwieldy).

In this chapter we have shown that logics for programming languages may contain features which mean that they cannot reasonably be coded using meta-binding and substitution. We have further shown that McKinna-Pollack binding provides an effective way to code these logics, but that in some cases the rules can become complex.

Another candidate method for coding logics for programming languages is the parallel substitution technique given by Stoughton [Sto88], mentioned in the introduction to this chapter. Parallel substitution would, we believe, give simple rules for parallel binding operators such as pattern matching. We know of no work in which parallel substitution is used for this purpose.

Chapter 5

Coding logics in logical frameworks

A *logical framework* is a formal system in which the rules and axioms of a logic may be given. There have been many formal systems proposed as logical frameworks (for example Feferman's FS0 [Fef94], the Edinburgh Logical Framework [AHM87]), and various mechanizations of these logical frameworks (for example LEGO [PL92]). Here we concentrate on Paulson's Isabelle [Pau94a], although our techniques could be used in most mechanizations of most logical frameworks. Isabelle was chosen simply for "engineering" reasons, rather than because of a philosophical bias: it is extremely easy to define new Isabelle logics (indeed, that is its *raison d'être*), and defining new logics for theorem provers is the subject of this chapter.

Isabelle is a generic theorem prover with simple and well-developed methods for coding logics and powerful automatic proof tools. Its logical framework is a simple form of constructive Higher-Order Logic where implication is used to code logical consequence and universal quantification is used to code eigenvariable conditions on rules.

In this chapter we continue the study of how best to represent logics for programs in logical frameworks. In the previous chapter we showed how complex patterns of binding could be represented; this chapter has two themes.

- In the first section we note that the usual methods of typechecking backwards proofs can be inefficient. To remedy this, we suggest a new method called *backwards typechecking*.

- In the second section we note that we can annotate goals with typechecking information, to further increase efficiency.

Many theorem provers have been designed to embody logical frameworks (Isabelle [Pau94a] and LEGO [PL92] for instance). Many others are used as logical frameworks (HOL [GM93] for instance). When discussing a logical framework we will often distinguish between the *meta-logic* in which logics are encoded (that is, the framework formal system itself), and the *object logic* (the logic which is encoded).

5.1 Backwards typechecking

In this section we develop a method for showing well-formedness conditions of judgements in backwards proof. A typical well-formedness condition is that a judgement is type-correct, but the method given here is quite general. To emphasize this fact, we begin by considering a general way of thinking about logics: consequence relations.

Definition 5.1 *Let $\mathcal{WFF}$ be a set of well-formed formulae. A relation $\Longrightarrow$ between finite sets of well-formed formulae is a consequence relation if it has the following properties:*

Weakening: *If $H \Longrightarrow C$, then $(H \cup H') \vdash C$.*

Cut: *If $H \Longrightarrow C$ and $(H' \cup C) \vdash C'$ then $(H \cup H') \Longrightarrow C'$*

Both Avron [Avr87] and Ryan and Sadler [RS92] give good introductions to consequence relations.

Now let us move on to consider judgements and well-formedness conditions. Suppose the set of well-formed judgements which we are considering is a subset $\mathcal{WFJ}$ of some larger set $\mathcal{J}$. For example $\mathcal{J}$ might be the set of syntactically correct judgments, and $\mathcal{WFJ}$ might be those judgements which are also type-correct. Let $\vdash$ be a consequence relation over $\mathcal{WFJ}$. We can, of course, also consider $\vdash$ to be a consequence relation over $\mathcal{J}$, since if a judgement is in $\mathcal{WFJ}$, it must be in $\mathcal{J}$. It is often much easier to generate $\mathcal{J}$ (it might be a free algebra represented as an ML datatype, say) than to generate $\mathcal{WFJ}$ (which requires evaluating the well-formedness condition in some way).

Automated proof assistants are slow: they can take seconds to prove even quite trivial theorems, and hours or days to prove large results. Logical frameworks exacerbate this problem since rules cannot be "hard-coded" for maximum efficiency. The central question which we shall address in this chapter is: how can we encode $\vdash$ in a logical framework such that the computation needed to ensure that a judgement is in $\mathcal{WFJ}$ is minimized?

Let us consider some ways to encode $\vdash$. The first is to ensure that every part of every rule only applies to well-formed judgements. Thus the rule $\frac{A \quad B}{C}$ would have the following side-conditions:

1. $A \in \mathcal{WFJ}$;
2. $B \in \mathcal{WFJ}$;
3. $C \in \mathcal{WFJ}$.

From the point of view of efficiency of computation, this could well be a disaster: checking for membership of $\mathcal{WFJ}$ occurs for every node and leaf in a proof tree. It might, for example, be the case that to verify side-conditions 1 and 2 has some computations in common with side-condition 3, but this system requires us to check all the side-conditions. One could argue that memoization would solve this problem, but A, B and C may be large, and memoization may use enormous amounts of memory for poor increases in speed.

Of course, formal systems are rarely designed using this method: clearly, having shown that $A \in \mathcal{WFJ}$ and $B \in \mathcal{WFJ}$, we do not need to re-perform the common computations in checking $C \in \mathcal{WFJ}$. This leads us on to the second way of encoding $\vdash$. We can characterize this method by the slogan "typechecking forwards":

- each axiom must be a well-formed judgement;
- whenever the antecedents of a rule are well-formed judgements, the consequent must be well-formed.

By induction on the size of derivation trees, it is clear that every node in a proof tree is a well-formed judgement.

The second coding method is much better than the first: we must only do enough checking to check that the consequent is well-formed, given that the hypotheses are. There are, however, two situations in which it is inefficient. Both occur during backwards proof, and since most machine-assisted proof is primarily backwards, this is a cause for concern.

- The first is when the proof tree has two or more identical (or similar) judgements at its leaves. In this case, checking each leaf for well-formedness will require some duplication of effort. One could, of course, record the duplicated well-formedness lemma and re-use it, but this means that we should record *every* well-formedness proof—this will be an enormous burden in time and space (not to mention programming).

- The second situation is when the judgement is not, in fact, well-formed. We may be able to continue the proof at great length: if the antecedents are ill-formed, we may be able to prove the necessary side conditions to show that the consequent is well-formed. Only at an advanced stage is it apparent that the proof can proceed no longer: we have been asked to show that some ill-formed judgement at a leaf of the proof tree is well-formed. There is nothing to do except to correct the initial goal and begin the proof again.

  This is obviously undesirable. One possible fix is to check that the original goal judgement is well-formed. But then we are duplicating effort if the goal is well-formed: the side conditions in the proof tree will guarantee that the original goal is well-formed, why must we prove it again?

We are now in a position to suggest an improved method of coding. Since proof is being done backwards (from the root of the proof tree out to the leaves), it is more natural, more convenient and more efficient to check well-formedness backwards too. The slogan now is "backwards typechecking."

- Show that the initial goal is well-formed.

- Whenever the *consequent* of a rule is well-formed, the *antecedents* must be well-formed judgements.

At first this seems rather odd: one is naturally used to proof rules which deduce the consequent from the antecedents. However, it is again simple to see by induction that for such a system every node in a proof tree must be a well-formed judgement.

Now let us consider the initial goal in more detail. How can we ensure that the initial goal is well-formed? We split the formal system into two types of judgements.

1. A judgement type for initial goals, $\vdash_1$ say, which we require to be well-formed.

2. A judgement type for the rest of the proof tree, $\vdash_2$ say: we require that if the consequent of a rule in this system is well-formed, the antecedents must also be well-formed judgements.

The only rule we need to connect these judgement types is as follows:

Well-formedness introduction (`WFI`):

$$\frac{\vdash_2 P}{\vdash_1 P}$$

- $P \in \mathcal{WFJ}$.

The proof system proper resides in deduction rules for $\vdash_2$. If we have managed to prove a $\vdash_1$ goal, we must first have applied `WFI` (and proved the goal is a well-formed judgement). The rest of the proof has used the rules for $\vdash_2$, where the rules preserve well-formedness backwards, and so the entire proof tree is well-formed.

Turning our attention to the $\vdash_2$ proof system, let us look in particular at axioms. Axioms are rules without antecedents: it is vacuously true that their antecedents are well-formed. Thus axioms *never* need well-formedness conditions. Even ill-formed judgements can be axioms: the point is that starting from a well-formed goal, one can never reach ill-formed axioms.

When one is deriving rules, the natural place to do it is in the $\vdash_2$ system, since that is where proofs happen. When deriving theorems, however, they should be in the $\vdash_1$ system: that is the only way which guarantees their well-formedness.

It would be quite sound to add rules to extract information from the $\vdash_1$ judgement. There are essentially two sorts of information that we can get from a $\vdash_1$ judgement.

1. The judgement is well-formed.

   Well-formedness (`WF`):
   $$\frac{\vdash_1 P}{P \in \mathcal{WFJ}}$$

2. Every member of $\mathcal{WFJ}$ is a member of $\mathcal{J}$, so every $\vdash_1$ theorem is a $\vdash_2$ theorem.

   Well-formedness elimination (`WFE`):

   $$\frac{\vdash_1 P}{\vdash_2 P}$$

There seems to be little utility to such rules in most cases, (the rule `WF` is probably the more useful). We omit them.

We have not yet accounted for rules with discharged hypotheses. How should they be represented in $\vdash_2$? Given a well-formed consequent, not only must the antecedents be well-formed, but any discharged hypotheses must be too.

Finally, note that it is not particularly novel to ensure that well-formedness conditions hold during backwards proof in this way. Many theorem provers check type-correctness as soon as a goal is entered. The novelty of this section lies in two facts:

- we make the necessary encoding explicit, rather than hiding it or ignoring it or even failing to recognize that such an encoding has been made;
- we give a theoretical account of such an encoding, rather than proceeding informally and relying on some sort of intuition to show that the logic actually ensures well-formedness in the desired way.

The advantage of this is the advantage of formality versus lack of rigour. The correctness of the method, and of its particular instances, are now open to mathematical enquiry and criticism.

5.2 Encoding context within a logic

In the previous section we showed how to encode a logic in logical framework in order to make backwards proof more natural and more computationally efficient. In this section, we simply make the observation that, in a logic with a well-formedness condition coded in this way, proofs may still repeat some work (for example, constructing a typechecking context). In some such cases, we may be able to encode some contextual information in the judgement, and prevent its repeated calculation.

Let us replace $\vdash_2$ with another judgement form: $\vdash_C P$, where C is some annotation derived from P in order to make checking of well-formedness easier (for example, it might include a type environment to make type checking easier). The judgement $\vdash_C P$ then means "if C is the annotation for P, and P is well-formed, then P holds."

The rule `WFI` then changes.

Well-formedness introduction (`WFI`):

$$\frac{\vdash_C P}{\vdash_1 P}$$

- C is the annotation for P;
- Using C, we can show $P \in \mathcal{WFJ}$.

This rule does precisely the same work as the previous version, but it splits the well-formedness check in two. In the first side-condition, we extract the annotation C from P. In the second, we use this annotation to show well-formedness.

Rules in $\vdash_C$ have the following properties.

- If the annotation for the consequent is correct, the annotations for the antecedents must be correct.
- If (using the annotation) the consequent is well-formed, then (using the annotation) the antecedents are well-formed.

It is in these rules that we are saving some work. Having annotation for the consequent, we need to get annotation for the antecedents. This might be very easy: the antecedents' annotations might be exactly the same as the consequent's. In the system $\vdash_2$ we would have to have built the annotations for the antecedents from scratch.

On the other hand, the annotation for the consequent might not tell us anything about the annotation for the antecedents, and we might need to extract the annotation for the antecedents from scratch. In this case we do not gain anything over $\vdash_2$ by using $\vdash_C$, but then again neither have we lost anything, since we would have to calculate all the annotations as part of the well-formedness checks in $\vdash_2$ anyway.

The precise encoding of the logic in the logical framework could well arrange for annotations to be hidden from the user during proof. This would mean that, although the rules of the logic would be annotated, proofs in the logic would, for the sake of simplicity, appear not to be.

It should be noted that deciding on which annotation will be most useful is an art rather than a science. More information will presumably make well-formedness checks easier, but may well obscure the meaning of the rules with large numbers of side-conditions relating to the form of the annotation.

Finally, notice that again the suggested encoding is not particularly novel. Many theorem provers maintain context information for a judgement. The novelty of this section is once more that we make the context information explicit within the judgements of the formal system, rather than maintaining the context deep within the code of the theorem prover. And once more the advantage is that the method of maintaining context information is formalized and is open to enquiry and criticism.

Part II

Core ML

Chapter 6

Simplified ML

> At a time when it is increasingly understood that programs must withstand rigorous analysis, particular for systems where safety is critical, a rigorous language presentation is even more important for negotiators and contractors; for a robust program written in an insecure language is like a house built upon sand.
>
> —Robin Milner, Mads Tofte and Robert Harper,
> *The Definition of Standard ML*

Standard ML is a very rich language—too rich to allow in a single dissertation the kind of investigation I make here of a part of it. It is necessary, then, to make simplifications, to reduce ML to a language with much of the essential flavour of Standard ML, but which is much smaller. I call this language Simplified ML. In this chapter, I describe Simplified ML, and in later chapters I discuss how to reason about it.

I discuss the Core part of Simplified ML in some detail. I first describe it informally and discuss its features, and then define it formally with respect to the Definition (of *Standard* ML).

In this dissertation, the detailed definition of the Modules part of Simplified ML will be of less importance than the definition of the Core language. It is therefore discussed more briefly.

We refer to the 1990 Definition of Standard ML [MTH90] throughout. However, the changes in the 1997 Definition would have little impact on the work in this dissertation.

6.1 The Core Language

6.1.1 Changes from Standard ML

In this section I informally describe the changes made to turn Standard ML into Simplified ML, at the Core language level. Here are the omissions and changes (roughly in order of their significance):

- The primary simplification is that I choose to reason only about purely functional programs, and to exclude assignment to (mutable) store.
- Simplified ML has no exception mechanism.
- Functions are all total—this totality is enforced by a simple syntactic check on function declarations (described in Section 6.1.2).
- Datatype value constructors cannot take function arguments in which elements of the type being declared occur.
- Declarations of datatypes must give the datatype a unique, explicit type name (in addition to the type constructor).
- Patterns are non-overlapping and exhaustive.
- ML function abstraction is only allowed in the form `fn` *var* `=>` *exp*, and not in the more general form `fn` *match*. Pattern matching is provided by the `case` construction.
- Similarly, value bindings must be of the form *var* `=` *exp*, not the form *pat* `=` *exp*.
- Simultaneous declaration of datatypes is not allowed, nor is simultaneous declaration of values.
- Declarations of free variables (`param` declarations) are added.
- There are no "basic values", except the equality test: all other functions must be declared using the mechanisms provided within Simplified ML. This means that the initial basis cannot contain functions with side-effects, such as `input`, and there are no overloaded functions.
- There is no `type` declaration.
- There are no layered patterns.

- There is no wildcard pattern (_), and no wildcard pattern row (...).
- Sequential composition of declarations forms declaration sequences (new syntactic classes, FDecSeq and BDecSeq), rather than trees of declarations (syntactic class Dec).
- Type constraints are not allowed.
- There are no fixity declarations.
- Simplified ML has no `abstype` declaration.

Core Simplified ML, then, is similar to the subset of ML used in LAMBDA 4.0. It does differ in some important ways, however—for example, LAMBDA `let` expressions are not as general as Simplified ML `let` expressions, and are not polymorphic, as Simplified ML `let` expressions are, and LAMBDA has no need of a `local` declaration.

Some of these restrictions are made in order to ensure that the logic can be sound (for example, allowing only total functions); some of the restrictions are made because the omitted features add little functionality, but complicate the language considerably (no overlapping patterns); some are made for other reasons. Many of them require some further comment.

There are two reasons for excluding state and assignment. The first is pragmatic: reasoning about programs with state is a hard problem,and would have occupied virtually all my attention. One of the problems is that including state destroys "referential transparency"—the ability to substitute like for like in a program. The problem is that two program fragments can return the same value, and yet modify the state in different ways. The second reason is "theological": once assignment is added ML becomes, for our purposes, just another imperative language—I believe that functional programs are easier to write and verify than imperative programs, and so I have restricted my study to the purely applicative part of Standard ML.

Exceptions are also difficult to reason about, and can also destroy referential transparency:

```
exception E1 and E2

(fn x => fn y => (x, y)) (raise E1) (raise E2)
```

This expression raises the exception `E1`, because of the order in which it is evaluated. But consider the following expression:

```
(fn y => fn x => (x, y)) (raise E2) (raise E1)
```

In Standard ML, this raises the exception `E2`. In an exception-free language, the following expressions are provably equal:

`(fn x => fn y => (x, y))` e_1 e_2
`(fn y => fn x => (x, y))` e_2 e_1

In the absence of exceptions, one can sometimes make use of a passable substitute by using datatypes to give a disjoint union type, such as Berry's `('a, 'b) Result` type [Ber91], although this approach quickly becomes unworkable in the face of anything other than the simplest applications. I choose to omit exceptions from Simplified ML, although I recognize that to do so is to ignore an important part of ML.

It can be argued, (for example by Hofmann [Hof92]) that restricting a logic for a programming language to total functions makes little practical difference:

- almost all carefully-written programs are terminating;
- those programs which do not terminate (for example, the read-eval-print loop of a LISP interpreter) are constructed from terminating functions in a simple way (`print` $\circ$ `eval` $\circ$ `read` put into an endless loop).

In only reasoning about terminating functions, one does not significantly reduce understanding of the few non-terminating examples. Whilst I do not find this argument completely convincing, since it rather glosses over how one is supposed to prove anything about a non-terminating program from a proof about a terminating program, it does have some merit, since most programs do terminate. There are, of course, honourable exceptions: Knuth-Bendix completion is an example of a method about which one might like to reason, but it is a semi-algorithm, and so cannot be coded in Simplified ML.

The principal reason for excluding non-terminating functions is to allow Simplified ML to be embedded in a well-understood logic (a version of Higher-Order Logic [Chu40]). Cheng and Jones [CJ91] give a useful survey of the field of logics of partial functions—although this does not include some recent developments: for example, Finn, Fourman and Longley's work outlined later. One approach to partial functions is that of LAMBDA 3.2 [FF90], which is an implementation of Fourman and Scott's Logic of Partial Elements [Sco79], which has a great advantage over our logic in that there is no termination check on function declarations—this check excludes the most natural definitions of some functions which are total, and is quite artificial. By using this logic one quickly becomes bogged down in proving existence conditions—as a rough guide to the severity of this problem, Goossens [Goo93] estimates that 60% of the list permutations done whilst using LAMBDA 3.2 involve the list of existence conditions.

If the logic is to be restricted to terminating functions, how are functions to be proved total? A well-known method of proof is to give a measure of complexity, in a well-ordered type, of a function expression and then to show that this complexity strictly decreases in recursive calls of the function. If the user were expected to prove every function total by hand, she would quickly become swamped by these proof obligations (in a similar, but less serious, fashion to the proof of existence conditions in LAMBDA 3.2). Instead, functions in Simplified ML are shown to be total by a simple syntactic check. This check is a special case of the "complexity" argument, where the complexity is the lexicographic product of the "sizes" of the patterns in the recursive call (this is similar to a scheme proposed by Burstall [Bur87]). All functions which pass this check are total, but many total functions fail the check. A better system would allow the user to prove termination of a function by hand in the case that the syntactic check failed but the user could see that the function was, indeed, terminating. Although this is quite feasible in theory, it makes implementation more troublesome and has not been adopted in the current prover.

Another possibility is to regard partial functions as loose specifications of total functions—the function declaration still specifies a total function, but one can only reason about the restriction of the function to that part of its domain over which its value is defined by its declaration. This idea is described by Finn, Fourman and Longley [FFL97]. It has the advantage that once a function is proven total, one need no longer reason about its definedness, (making it no more complex than the system we adopt for total functions), but in addition one may prove properties of partial functions. However, the current version of the scheme is a poor match with ML—some total functions cannot be proven total, and, more seriously, some functions which diverge can be proven to have a value (a similar problem occurs in our logic unless great care is taken over substitution, see Section 7.3.3).

Datatype declarations cannot have constructors which take function types containing the type being declared for two reasons:

- The type cannot appear on the left of a function type (a *negative* occurrence) because one could not then model the type as a set—it is too large. (Such declarations are valid when types are modelled as domains, but this is irrelevant for our models).

- The type cannot appear on the right of a function type because this means that it is impossible (to my knowledge) to give an induction rule for the type.

Datatypes in Standard ML are generative (that is, two identical `datatype` declarations give rise to different types). During elaboration (Rule 29), each datatype is given a unique type name, which the programmer never sees, but which is used to tell datatypes apart. It is necessary to make these names explicit in Simplified ML. Consider the judgement:

$$\mathit{decseq} \vdash_C \texttt{let datatype } T \texttt{ = } c \texttt{ in } c \texttt{ end}$$
$$\approx \texttt{let datatype } T \texttt{ = } c \texttt{ in } c \texttt{ end}$$

There are two interpretations of this:

- it is a straightforward instance of the reflexivity rule, and is valid;
- the datatypes must be different because of generativity, hence the judgement cannot be valid: there must be a hidden difference between the datatypes (this difference is between the type names).

These interpretations are contradictory, but both may be appropriate at different times. The problem lies in the fact that the proposition part of a judgement is not ordered in the same way as a declaration sequence—one cannot simply conclude that two datatype declarations refer to different types.

One solution might be to abandon generativity, and identify all datatypes which have identical declarations. But one could then have judgements such as:

$$\ldots;\texttt{datatype } T \texttt{ = } c;\ldots;\texttt{val } x \texttt{ = } c;\texttt{datatype } T \texttt{ = } c \vdash_C$$
$$x \approx c$$

This seems to run counter to the ideas of ML—one must look through all previous declarations in order to ensure that a datatype is new.

It is necessary, then, that datatypes have a type name associated with them. The type name of a type is used whenever a type constructor or value constructor of that type is used. It would be possible for a system to annotate declarations and type and value constructors with type names automatically. Type names should be unique (although, as we shall see in Chapter 7, the logic does not enforce this restriction).

Simplicity is the principal reason why patterns in a match must be non-overlapping—if they were to overlap it would be necessary to add conditions to any rule reasoning about the match to the effect that no previous patterns matched successfully. Such conditions would be enormously complex, and so allowing overlapping patterns would hinder rather than help the user. Expressions containing non-exhaustive patterns may have no simple interpretation in the model we adopt—what value does an expression denote when pattern matching fails?

Declarations of free variables are necessary for reasoning about λ abstractions and quantifiers. Type constraints are omitted in order to simplify the set U of scoped type variables in a context.

All other changes are made in order to reduce the language to a manageable size.

6.1.2 Syntax

The grammar for Core Simplified ML is given in Figures 6.2, 6.1 and 6.3—compare with Figure 3 on page 8 of the Definition.

The Definition's syntax for "*...seq*" has been modified in the case of the syntactic classes BDecSeq and FDecSeq, so a *bdecseq* is a semi-colon separated list of declarations (the semi-colon acts like `cons`), and an *fdecseq* is a semi-colon separated list of declarations running in the other direction (the semi-colon acts like `tack`), whereas the Definition says a *decseq* is either a single declaration or a comma-separated list of declarations enclosed in parentheses. The empty declaration sequences in both BDecSeq and FDecSeq is written `nothing`. This difference between the two types of declaration sequence is simply in order to make the formulation of certain rules more natural, for example `let` introduction (`LetI`), and splitting `let` expressions (`LetCong2`).

Derived forms are treated exactly as in Standard ML. Simplified ML has all the derived forms of Standard ML, and in addition singleton declaration sequences may be written as simply *dec* rather than `nothing;` *dec* or *dec*`; nothing`. This means that the constant `nothing` need never explicitly appear except when a declaration sequence with zero items is required: all other declaration sequences can be formed by adding items to singleton lists. There is also an append operation on both forms of declaration sequence, written $decseq_1$ `@` $decseq_2$.

In this section, I treat the difference in their method of formation as the only difference between the two kinds of declaration sequence. This glosses over an important point: our representation of the logic has two different kinds of variable—*b-variables* and *f-variables*, so a *bdecseq* binds b-variables to declarations and an *fdecseq* binds f-variables to declarations. In the grammar, all declaration sequences are simply written *decseq*. Declarations must be parametrised on the type of variable bound—but both sorts of declaration have the same syntax, so here we denote each simply by *dec*. The distinction between b-variables and f-variables is explained at some length in Chapter 4, and the use of each is clarified there: here, they are assumed to be the same.

There are some restrictions on datatype declarations. Given a declaration:

$$\texttt{datatype}\ \mathit{tvs}\ \mathit{tc}_{tn}\ \texttt{=}\ \mathit{conbind}$$

the datatype is well-founded (i.e. non-empty) if there is at least one non-recursive value constructor. Testing this is quite simple (the encoding is omitted). It is also necessary to test if a datatype is well-formed in the sense that recursive occurrences in the declaration of the value constructors are identical to (have the same type-variable parameters as) the binding occurrence:

```
datatype ('a, 'b) T__t1 = E | N of 'a * 'b * ('a, 'b) T__n
```

satisfies this criterion, but

```
datatype ('a, 'b) T__t1 = E | N of 'a * 'b * ('b, 'a) T__n
```

does not, since `('a, 'b) T__n` is different to `('b, 'a) T__n`.

All Simplified ML datatype declarations must satisfy this condition and must be well-founded. In addition, the datatype being declared may not appear as part of a function type in its own declaration.

Given a `case` expression of the form `case` e `of` p_1 `=>` e_1 `|` ... `|` p_m `=>` e_m (this may be a RecCase expression), if p_i matches e, then there is no other j, $j \neq i$, such that p_j matches e—that is, patterns are exclusive—and there will always be an i such that p_i matches e—that is, patterns are exhaustive.

Given a recursive function declaration

$$\mathit{recdecf}\,\texttt{fn}\ v_1\ \texttt{=>}\ \ldots\texttt{=>}\ \texttt{fn}\ v_n\ \texttt{=>}\ \texttt{case}\ \{\mathit{exprow}\}\ \texttt{of}\ \mathit{match}$$

then *exprow* must be $\texttt{1} = v_1, \ldots, \overline{n} = v_n$, where $\overline{n}$ is the numeral representing n. As an aside, when *match* is:

$$\begin{array}{ll} \{\texttt{1} = p_{1,1}, \ldots, \overline{n} = p_{1,n}\} & \texttt{=>} e_1\ | \\ \vdots & \\ \{\texttt{1} = p_{m,1}, \ldots, \overline{n} = p_{m,n}\} & \texttt{=>} e_m \end{array}$$

this exactly corresponds to the derived form:

$$\begin{array}{lll} \texttt{fun} & f\ p_{1,1} \ldots p_{1,n} & = e_1 \\ & \vdots & \\ | & f\ p_{m,1} \ldots p_{m,n} & = e_m \end{array}$$

$$
\begin{array}{rcl}
\mathit{atexp} & ::= & \mathit{var} \\
 & & \mathit{con}_{\mathit{tyname}} \\
 & & \{\ \langle \mathit{exprow} \rangle\ \} \\
 & & \texttt{let}\ \mathit{decseq}\ \texttt{in}\ \mathit{exp}\ \texttt{end} \\
 & & \texttt{(}\ \mathit{exp}\ \texttt{)} \\
 & & \\
\mathit{exprow} & ::= & \mathit{lab}\ \texttt{=}\ \mathit{exp}\ \langle\ \texttt{,}\ \mathit{exprow} \rangle \\
 & & \\
\mathit{exp} & ::= & \mathit{atexp} \\
 & & \mathit{exp}\ \mathit{atexp} \\
 & & \texttt{fn}\ \mathit{var}\ \texttt{=>}\ \mathit{exp} \\
 & & \texttt{case}\ \mathit{exp}\ \texttt{of}\ \mathit{match} \\
 & & \\
\mathit{match} & ::= & \mathit{mrule}\ \langle\ \texttt{|}\ \mathit{match} \rangle \\
 & & \\
\mathit{mrule} & ::= & \mathit{pat}\ \texttt{=>}\ \mathit{exp}
\end{array}
$$

Figure 6.1: The grammar for Simplified ML expressions and matches

Recursive functions must be in a form which guarantees them to be total. Given some recursive occurrence of the function being declared, $f\ a_1\ \ldots\ a_m$, where f can have at most n arguments, $n \geq m$, this occurrence must, because of the conditions on recursive function declarations, be in a match rule of the form:

$$(p_1, \ldots, p_n)\ \texttt{=>}\ e$$

For each such occurrence of f, there is a k, $1 \leq k \leq m$, such that for $1 \leq i < k$, each a_i is exactly p_i, and a_k is strictly smaller than p_k.

The relation "smaller" is the smallest relation satisfying:

- an expression is smaller than a pattern which is identical to it;
- if e is smaller than p, then e is smaller than p', where p' is a pattern containing p as a sub-pattern.

An expression is strictly smaller than a pattern if it is smaller and is not identical to the pattern (this definition is equivalent to the one given by Coquand [Coq92]).

In addition to these changes, the Isabelle parser generates abstract syntax trees in which the syntactic classes Exp and AtExp are joined. It also converts *reccase* to the corresponding `case` expression, and *recexp* to `fn` expressions.

$$
\begin{array}{rcl}
\mathit{dec} & ::= & \texttt{val}\ \mathit{valbind} \\
 & & \texttt{datatype}\ \mathit{datbind} \\
 & & \texttt{local}\ \mathit{decseq}_1\ \texttt{in}\ \mathit{decseq}_2\ \texttt{end} \\
 & & \\
\mathit{bdecseq} & ::= & \texttt{nothing} \\
 & & \mathit{dec}\ \texttt{;}\ \ \mathit{bdecseq} \\
 & & \mathit{dec} \\
 & & \\
\mathit{fdecseq} & ::= & \texttt{nothing} \\
 & & \mathit{fdecseq}\ \texttt{;}\ \ \mathit{dec} \\
 & & \mathit{dec} \\
 & & \\
\mathit{valbind} & ::= & \mathit{var}\ \ \texttt{=}\ \ \mathit{exp} \\
 & & \texttt{rec}\ \mathit{recbind} \\
 & & \\
\mathit{recbind} & ::= & \mathit{var}\ \ \texttt{=}\ \ \mathit{recexp} \\
 & & \\
\mathit{recexp} & ::= & \texttt{fn}\ \mathit{var}\ \texttt{=>}\ \mathit{recexp} \\
 & & \texttt{fn}\ \mathit{var}\ \texttt{=>}\ \mathit{reccase} \\
 & & \texttt{(}\ \mathit{recexp}\ \texttt{)} \\
 & & \\
\mathit{reccase} & ::= & \texttt{case}\ \{\ \mathit{exprow}\ \}\ \texttt{of}\ \mathit{match} \\
 & & \\
\mathit{datbind} & ::= & \mathit{tyvarseq}\ \mathit{tycon}_{\mathit{tyname}}\ \ \texttt{=}\ \ \mathit{conbind} \\
 & & \\
\mathit{conbind} & ::= & \mathit{con}\ \langle \texttt{of}\ \ \mathit{ty}\rangle\ \langle \texttt{|}\ \ \mathit{conbind}\rangle
\end{array}
$$

Figure 6.2: The grammar for Simplified ML declarations and bindings

$$
\begin{array}{rcl}
atpat & ::= & var \\
& & con_{tyname} \\
& & \texttt{\{}\ \langle patrow \rangle\ \texttt{\}} \\
& & \texttt{(}\ pat\ \texttt{)} \\[1em]
patrow & ::= & lab\ \texttt{=}\ pat\ \langle \texttt{,}\ patrow \rangle \\[1em]
pat & ::= & atpat \\
& & con_{tyname}\ atpat \\[1em]
ty & ::= & tyvar \\
& & \texttt{\{}\langle tyrow \rangle\texttt{\}} \\
& & tyseq\ tycon \\
& & ty_1\ \rightarrow\ ty_2 \\
& & \texttt{(}\ ty\ \texttt{)} \\[1em]
tyrow & ::= & lab\ \texttt{:}\ ty\ \langle \texttt{,}\ tyrow \rangle
\end{array}
$$

Figure 6.3: The grammar for Simplified ML patterns and type expressions

6.1.3 Static semantics

Rules which are concerned with features omitted from Simplified ML are omitted completely—they are given in Table 6.1. If a rule has an alternative form, the options are given letters, so a rule $x\langle y\rangle\langle\langle z\rangle\rangle$ has options (a) x, (b) xy, (c) xz, and (d) xyz.

Rule 6 should be changed to reflect the fact that the correct syntax for `let` expressions is now `let` *decseq* `in` *exp* `end` and not `let` *dec* `in` *exp* `end`. Similarly, Rule 22 should reflect the new syntax `local` $decseq_1$ `in` $decseq_2$ `end`, which replaces `local` dec_1 `in` dec_2 `end`.

Rule 24 now refers to the empty declaration *sequence* and not the empty declaration, and Rule 25 should reflect the fact that there are two forms of sequential composition: one acting as `cons` on b-declaration sequences, the other acting as `tack` on f-declaration sequences.

There are also some changes to be made to Rule 27.

$$\frac{C + VE \vdash \mathrm{ToValBind}(recbind) \Rightarrow VE}{C \vdash \texttt{rec}\ recbind \Rightarrow VE}\ 27$$

The function ToValBind is the obvious injection from the syntactic class RecBind to the Standard ML syntactic class ValBind.

Rule number	Description
1	special constants
4	exception constructors
11	type constraints
12	`handle` expressions
13	`raise` expressions
18	`type` declarations
20	`abstype` declarations
21	`exception` declarations
23	`open` declarations
25(a)	sequential composition without ;
26(b)	simultaneous value bindings
28	type bindings
29(b)	simultaneous datatype bindings
31	exception bindings
32	exception bindings
33	wildcard pattern
34	special constant patterns
37	exception constant patterns
40	labelled record wildcard pattern
44	exception application patterns
45	type constrained patterns
46	layered patterns

Table 6.1: Rules omitted from the static semantics of Simplified ML

6.1.4 Dynamic semantics

Since Simplified ML has neither exceptions nor imperative assignment, both the exception convention and the state convention are irrelevant, and we take the rules to be just those printed in the Definition. Rules which are concerned with features omitted from Simplified ML are omitted completely—they are given in Table 6.2.

There are a number of minor alterations to the remaining rules similar to the alterations made in the static semantics.

6.2 The Modules Language

In the previous section, we described in detail the modifications needed to turn Core Standard ML into Core Simplified ML. We approach the Modules language in a different way: we describe the properties we require of a modules system, and, without going into detailed definitions, observe that the ML Modules language needs only minor changes in order to have these properties.

- Every module must have an explicit signature. This means that we need not worry about the difference between principal and non-principal signatures. Obviously, one could (automatically) annotate Standard ML Modules programs with principal signatures, so one can regard this requirement as largely cosmetic.

- Every visible value must have a visible type. Consider the following example.

  ```
  structure S =
  struct
    datatype T = C;
    val x = C;
    datatype T = C;
  end;
  ```

  The type of `x` is not visible. There appear to be no real uses for this "feature," and I contend that no well-written program uses it.

- We do not consider functors in this dissertation.

Rule number	Description
103	special constants
106	exception constructors
114	`ref` application
115	`:=` application
116	basic value applications
118	pattern matching failure in applications
120	`handle` expressions
121	`handle` expressions
122	`raise` expressions
125	pattern matching failure in matches
126	pattern matching failure in matches
128	pattern matching failure in match rules
130	`exception` declarations
132	`open` declarations
134(a)	sequential composition without ;
135(b)	simultaneous value bindings
138	exception bindings
139	exception bindings
140	wildcard pattern
141	special constant patterns
142	special constant patterns
145	pattern matching failure in constructor constant patterns
146	exception constant patterns
147	exception constant patterns
150	labelled record wildcard pattern
151	pattern matching failure in labelled record patterns
155	pattern matching failure in constructor application patterns
156	pattern matching failure in exception application patterns
157	pattern matching failure in exception application patterns
158	`ref` application patterns
159	layered patterns

Table 6.2: Rules omitted from the dynamic semantics of Simplified ML

Chapter 7

Reasoning about Core Simplified ML

> In a word, he would say, error was error,—no matter where it fell,—whether in a fraction,—or a pound,—'twas alike fatal to truth, and she was kept down at the bottom of her well as inevitably by a mistake in the dust of a butterfly's wing,—as in the disk of the sun, the moon, and all the stars of heaven put together.
>
> He would often lament that it was for want of considering this properly, and of applying it skilfully to civil matters, as well as to speculative truths, that so many things in this world were out of joint;—that the political arch was giving way;—and that the very foundations of our excellent constitution in church and state, were so sapped as estimators had reported.
>
> —Laurence Sterne, *Tristram Shandy*

In this chapter I give a logic, called *Emily*, for reasoning about Core Simplified ML. This logic is designed with implementation in Isabelle in mind. It contains a version of Higher-Order Logic similar to the logic given in Chapter 4. The features of note are:

- it uses McKinna-Pollack binding, as does the logic of Chapter 4;
- it uses backwards type-checking, and annotated deductions;
- in a judgement such as $\Gamma \vdash P$, we now allow Γ to contain Simplified ML declarations, and P to contain Simplified ML expressions.

Thompson and Hill [HT95] give a method for writing Miranda programs as Isabelle logics which has much in common with my work. (In comparing them, it should be borne in mind that Miranda is a lazy language, and so many of the rules of the logic are different to the corresponding rules for ML; for example, the Miranda substitution rule need not be careful to avoid situations which cause non-termination in an eager language like ML. They also make different choices to me in other areas: they allow non-termination, for instance). Thompson and Hill give a *separate* logic for each Miranda program, and regard the process of producing the logic as a form of compilation. An example of an advantage of this style is that they can represent each Miranda type as a separate Isabelle type, and use Isabelle meta-type polymorphism to do the duty of Miranda type polymorphism (this is reminiscent of the advantages of shallow embeddings [BGG$^+$92]). The main disadvantage is that it is difficult to relate the properties of different Miranda programs, except by merging the theories representing them.

In contrast, I give a single logic for reasoning about *all* Simplified ML programs. The costs of this are reminiscent of the disadvantages of deep embeddings: there is one Isabelle type of ML types, and ML type inference must be encoded in all its detail. In fact, this encoding in Isabelle has been done by Cant and Ozols [CO92], (and an implementation of Emily in Isabelle would be free to use this). ML type inference is a complex process, and the fact that it has been encoded in this way means that we can use the exact ML type system in all its detail, rather than a compromise system chosen to fit easily with Isabelle typing.

However, the crucial advantage of the approach I take is that it is very easy to relate separate pieces of ML code, since each judgement can be relative to a different set of ML declarations. The central concern in this dissertation is reasoning about modular programs and the relation between them: this is made considerably easier by our ability to reason about different programs in the same derivation.

The Emily logic also has similarities to the logic of LAMBDA 4.0 [FF91], which allows reasoning about terminating programs in a functional subset of Core ML. As in Emily, the LAMBDA logic is based on the Simple Theory of Types. However, each ML program is represented in the LAMBDA prover as a separate object, and each proof is done relative to one such object. This means that LAMBDA has just the same drawbacks as the Miranda prover for reasoning about modular programs: programs cannot be related except by declaring them at the same time. A bonus of the form of judgements in Emily is that it becomes simple to deal with properly polymorphic `let` expressions: one can basically move declarations back and forth between the `let` expression and the context at will. In contrast, LAMBDA regards `let` as sugar for a λ abstraction and application, which means that each occurrence of a `let`-bound variable must have the same type.

Another benefit of this system is that `let` expressions can be treated in a correct, polymorphic fashion. This is because there are rules for moving declarations between the context of declarations and a `let` expression in the proposition. The expression `let val` v `=` e `in` e' `end` is not sugar for $(\lambda v.e')e$.

The exact formulation of Higher-Order Logic used in Emily is modelled on that used in HOL [GM93], with changes to account for the differences in the form of the logic. As is usual in Higher-Order Logic empty types are scrupulously avoided, unlike Paulson's formulation [Pau90] which embraces empty types. Terms of the logic may contain ML expressions (which may refer to the context of a judgement), and so the ML type system is a subset of the type system for the logic as a whole (in keeping with LAMBDA, and in contrast to usual Isabelle practice, purely logical types are referred to as *meta*-types).

It appears at the moment that there is little need for HOL's extensive type definition facilities—when a new type is needed, one simply declares an appropriate Simplified ML datatype. In turn, this makes the type of individuals largely redundant, and also the axiom of infinity. However, it may be that, as an extension to the system we give, one might wish to broaden the class of terminating ML functions allowed (recall that all Simplified ML functions must terminate). We might allow functions which were proved to terminate by the standard method of giving a measure of complexity—a value in a well-founded type—which strictly decreases with each recursive call. In this case, the possibility of describing well-founded types without resorting to ML may well be useful.

7.1 Syntax

The grammar for meta-types is given in Figure 7.1: it is important to notice that ML types form a subset of meta-types. The grammar for terms is given in Figure 7.2. Constants may not be re-bound by Simplified ML declarations, and a declaration which attempts to re-bind a constant is not well-formed. The constants have their usual derived forms; note that equality is typed: $M \approx_\tau M'$. Unlike the logic in Chapter 4, we treat $\wedge$, $\vee$, $\neg$, $\forall$, $\exists$, T and F as proper constants, rather than defining them at the start of the declaration sequence.

There are two judgement classes, each with a single form: their grammar is given in Figure 7.3. The syntactic class FDecSeq$'$ is similar to FDecSeq, except that it may contain declarations of free variables `param` *fvar*: *mty*. These *may not*

$$
\begin{array}{rcl}
mty & ::= & mtyvar \\
 & & ty \\
 & & mty \twoheadrightarrow mty
\end{array}
$$

Figure 7.1: The grammar for meta-types

$$
\begin{array}{rcl}
mterm & ::= & mvar \\
 & & exp \\
 & & \lambda mvar{:}\ mty.\ mterm \\
 & & mterm_1\ mterm_2 \\
 & & const \\
 & & \\
const & ::= & \supset \\
 & & \varepsilon \\
 & & \approx \\
 & & \wedge \\
 & & \vee \\
 & & \supset \\
 & & \neg \\
 & & \forall \\
 & & \exists \\
 & & \mathsf{T} \\
 & & \mathsf{F}
\end{array}
$$

Figure 7.2: The grammar for logical terms

$$
judge \quad ::= \quad fdecseq' \vdash mterm
$$

$$
tyjudge \quad ::= \quad fdecseq' \vdash_{context} mterm
$$

Figure 7.3: The grammar for judgements

appear in a declaration sequence anywhere else. They will be used to declare free variables, and this restriction simply means that they cannot appear inside "real" ML code.

A *context* is a semantic object, as described in the Definition. It is intended that in a system they would be transparent to the user: on input either *judge* goals are used, or Isabelle scheme variables which the system may instantiate with the appropriate value later; on output the system simply omits displaying contexts. Automatic tactics would handle the calculation and use of contexts as part of the elaboration process.

Type names, described in Section 6.1.1 have only one restriction on their use. Given types t_{tn} and $t'_{tn'}$ appearing anywhere in the same judgement, if tn and tn' are the same, then the type declarations must be the same. That is, the declarations must declare types and constructors with the same names and the same arities and argument types. Thus it is possible to declare identical types twice in a judgement, which is not possible in ML; however, it means that the reflexivity example is allowed:

$$
\begin{aligned}
ds \vdash_C & \texttt{let datatype T\_tn = C in C end} \\
& \approx_{\texttt{T\_tn}} \\
& \texttt{let datatype T\_tn = C in C end}
\end{aligned}
$$

7.2 Static Semantics of the Core Logic

In this section, I define the typing relation $C \vdash_{LOG} P \Longrightarrow mty$, where C is an ML type context, P is a term, and mty is a meta-type. The relation $C \vdash_{STAT} e \Longrightarrow ty$ is Simplified ML elaboration.

Meta-types are monomorphic. There is no facility for declaring meta-values in declaration contexts (although one could easily add it in the manner of Chapter 4), and `param` declarations are monomorphic. Although the logic as stated contains meta-type variables and the rule `InstType`, an Isabelle implementation could do away with these and instead use free Isabelle meta-variables to get typical ambiguity. Here is the rule `InstType`.

Type instantiation (`InstType`):

$$\frac{decseq \vdash_C P}{decseq[t/\alpha] \vdash_{C[t/\alpha]} P[t/\alpha]}$$

- No distinct type variables in P become identified after the substitution.

The HOL logic uses a simultaneous substitution operation to instantiate several types at once. A little thought makes it clear that the above rule is equivalent. The rule captures the behaviour of the relation $\succ$ between type-schemes in ML.

The basic type checking of terms is really extremely simple, thanks to the simple grammar for terms. We take advantage of the fact that terms never have any unbound b-variables (terms are *b-closed*) to reduce the task of typing variables to typing f-variables (if an expression is just a variable and is b-closed, then it must be an f-variable).

Meta-variables (`MVarElab`):

$$\frac{C(p) = t}{C \vdash_{LOG} p \Longrightarrow t}$$

ML expressions (`ExpElab`):

$$\frac{C \vdash_{STAT} \mathit{exp} \Longrightarrow t}{C \vdash_{LOG} \mathit{exp} \Longrightarrow t}$$

Abstractions (`AbsElab`):

$$\frac{C \vdash_{STAT} \texttt{param}\ p{:}\, t \Longrightarrow E \quad C \oplus E \vdash_{LOG} M[p/v] \Longrightarrow t'}{C \vdash_{LOG} \lambda v{:}\, t.\, M \Longrightarrow t \twoheadrightarrow t'}$$

Here, the `param` declaration is used simply to introduce a free variable of a certain type. LAMBDA's `vbl` declaration is similar, except that it is global in scope since LAMBDA has no notion of a declaration being local to a particular judgement.

Application (`AppElab`):

$$\frac{C \vdash_{LOG} M \Longrightarrow t' \ \twoheadrightarrow \ t \quad C \vdash_{LOG} N \Longrightarrow t'}{C \vdash_{LOG} M\ \ N \Longrightarrow t}$$

Constants (`ConstElab`):

$$\frac{t \in C_{INIT}(c)}{C \vdash_{LOG} c \Longrightarrow t}$$

In the initial context, constants have the following sets of types:

$$\begin{array}{rcl}
C_{INIT}(\varepsilon) & = & \{(A \twoheadrightarrow o) \twoheadrightarrow A, A \text{ is a meta-type}\} \\
C_{INIT}(\supset) & = & \{o \twoheadrightarrow o \twoheadrightarrow o\} \\
C_{INIT}(\approx) & = & \{A \twoheadrightarrow A \twoheadrightarrow o, A \text{ is a type}\} \\
C_{INIT}(\supset) & = & \{o \twoheadrightarrow o \twoheadrightarrow o\} \\
C_{INIT}(\wedge) & = & \{o \twoheadrightarrow o \twoheadrightarrow o\} \\
C_{INIT}(\vee) & = & \{o \twoheadrightarrow o \twoheadrightarrow o\} \\
C_{INIT}(\neg) & = & \{o \twoheadrightarrow o \twoheadrightarrow o\} \\
C_{INIT}(\forall) & = & \{(A \twoheadrightarrow o) \twoheadrightarrow A, A \text{ is a meta-type}\} \\
C_{INIT}(\exists) & = & \{(A \twoheadrightarrow o) \twoheadrightarrow A, A \text{ is a meta-type}\} \\
C_{INIT}(\mathsf{T}) & = & \{o\} \\
C_{INIT}(\mathsf{F}) & = & \{o\}
\end{array}$$

The type of constants is found from the *initial* context—since constants cannot be re-bound, one could as well use the current context. The meta-type o is the type of propositions.

Only one form of equality relation is needed for both logical terms and ML expressions, since meta-types include ML types as a subset; likewise, only one form of selection operator is needed.

It is necessary to add a form of declaration which declares free variables introduced by quantifier reasoning—this is the `param` declaration. These declarations are monomorphic, and so the following simple rule works.

$$\frac{C \vdash_{STAT} ty \Longrightarrow \tau}{C \vdash_{STAT} \texttt{param}\ p{:}\ ty \Longrightarrow \{p \mapsto \tau\}}$$

7.3 Deduction Rules

Let @@ be the append operation on declaration sequences. It will be useful to have the following definitions.

$$\begin{array}{rcl}
\mathsf{ExtCtxt}(C, \mathit{fdecseq}, C') & = & \exists E.\, C \vdash_{STAT} \mathit{fdecseq} \Longrightarrow E \wedge C' = C \oplus E \\
\mathsf{IsCtxt}(\mathit{fdecseq}, C') & = & \mathsf{ExtCtxt}(C_{INIT}, \mathit{fdecseq}, C') \\
\mathsf{ExtCorrect}(C, \mathit{fdecseq}, C', P) & = & \mathsf{ExtCtxt}(C, \mathit{fdecseq}, C') \wedge C' \vdash_{LOG} P \Longrightarrow o \\
\mathsf{IsCorrect}(\mathit{fdecseq}, C', P) & = & \mathsf{ExtCorrect}(C_{INIT}, \mathit{fdecseq}, C', P)
\end{array}$$

That is to say $\mathsf{ExtCtxt}(C, \mathit{fdecseq}, C')$ means that C' is the context obtained by elaborating $\mathit{fdecseq}$ in C. $\mathsf{IsCtxt}(\mathit{fdecseq}, C')$ means that C' is the context obtained by elaborating $\mathit{fdecseq}$ in the initial context. $\mathsf{ExtCorrect}(C, \mathit{fdecseq}, C', P)$ will be used to indicate that, given that $\mathit{fdecseq}' \vdash_C \ldots$ is type correct, then

$C' \vdash_{\mathit{fdecseq'}@@\mathit{fdecseq}} P$ is also type correct. $\mathsf{IsCorrect}(\mathit{fdecseq}, C, P)$ means that $C \vdash_{\mathit{fdecseq}} P$ is type correct.

It will be seen that there are cases in which this annotation fails to reduce the amount of type checking done. For example:

$$\frac{\mathit{fdecseq}@@\mathit{fdecseq'} \vdash_C Q}{\mathit{fdecseq} \vdash_{C'} P}$$

In rules such as this we must throw away the entire context C and rebuild C' from scratch. There are two ways around this.

- Change the form of elaboration contexts such that they are formed from a stack of "frames". When a rule such as the above is encountered, we can get to C' from C by popping the frames relating to *fdecseq'* from the stack. This has the disadvantage that ML elaboration contexts are defined in terms of sets: the notion of discarding frames is not used. Although frames are a standard notion in compilation, adopting them means diverging from the Definition. A more serious objection is that in some cases it may not be obvious how many frames should be discarded, and we may have to rebuild C' from scratch anyway: the more complex system would have gained us nothing.

- Annotate each declaration with the environment to which it elaborates. We can then quickly obtain C' by sticking together the environments for the declarations in *fdecseq*. This, however, increases the verbosity of rules and may also lead to a great increase in the space needed to store judgements.

Practical experience of the system is needed before we can judge if either of these two changes is justified. Such changes are about the level of detail of annotation provided, and are not crucial to the logical core of the system.

7.3.1 Datatype declarations and labelled record types

Each datatype has three rules associated with it:

- An induction rule. (This is the so-called "no junk" rule).

- A rule stating that terms formed from different constructors are distinct. ("No confusion").

- A rule stating that constructors are injective, that is to say, if two terms formed by applying arguments to the same constructor are equal then the arguments to the constructor are equal.

These rules are different for each datatype: sufficiently different that they cannot be expressed simply as instances of a schematic rule in Isabelle. There are, however, two ways in which these rules can be generated. Isabelle now has an "oracle" facility. This allows external reasoners to be applied to solve part of an Isabelle proof. It would be possible to write an ML program to create the rules for a datatype in the form required by Isabelle, and then to use this ML program as an oracle. This approach has the following features:

- the ML program would produce the rules quickly;
- the ML code would be easy to program, and comparatively clear;
- however, there is no mathematical definition of the rules, and, short of verifying the ML code, it is difficult to prove properties of the rules which would increase our confidence in their correctness.

Alternatively, the rules can be generated directly via a "logic program" (expressed as a primitive recursive function, or an inductive set) written as part of the Isabelle theory. For example, here is the induction rule for a datatype in Isabelle:

Datatype induction (`Ind`):

$$\frac{}{\mathit{decseq};\ \texttt{datatype}\ \mathit{tvs}\ \mathit{tc}_{\mathit{tn}}\ \texttt{=}\ \mathit{conbind} \vdash_C R}$$

- $\mathsf{Ind}(\mathit{tvs}, \mathit{tc}, \mathit{tn}, \mathit{conbind}, R)$

The relation Ind is itself given by Isabelle rules, of which these few will give a flavour:

$$\frac{\mathsf{Ind1}(\mathit{tvs}, \mathit{tc}, \mathit{tn}, \mathit{conbind}, p, R[p/P])}{\mathsf{Ind}(\mathit{tvs}, \mathit{tc}, \mathit{tn}, \mathit{conbind}, \forall P\colon \mathit{tvs}\ \mathit{tc}_{\mathit{tn}} \twoheadrightarrow o.\ R)}$$

- $p \notin R$

$$\frac{}{\mathsf{Ind1}(\mathit{tvs}, \mathit{tc}, \mathit{tn}, \mathit{con}\ c, p, (p\ c) \supset \forall e\colon \mathit{tvs}\ \mathit{tc}_{\mathit{tn}}.\ p\ e)}$$

$$\frac{\mathsf{Hyp}(\mathit{tvs}, \mathit{tc}, \mathit{tn}, c, t, p, H)}{\mathsf{Ind1}(\mathit{tvs}, \mathit{tc}, \mathit{tn}, \mathit{con}\ c\ \texttt{of}\ t, p, H \supset \forall e\colon \mathit{tvs}\ \mathit{tc}_{\mathit{tn}}.\ p\ e)}$$

$$\frac{\mathsf{Ind1}(\mathit{tvs}, \mathit{tc}, \mathit{tn}, \mathit{conbind}, p, R)}{\mathsf{Ind1}(\mathit{tvs}, \mathit{tc}, \mathit{tn}, \mathit{con}\ c\ \texttt{|}\ \mathit{conbind}, p, (p\ c_{\mathit{tn}}) \supset R)}$$

$$\frac{\mathsf{Hyp}(tvs, tc, tn, c, t, p, H) \quad \mathsf{Ind1}(tvs, tc, tn, conbind, p, R)}{\mathsf{Ind1}(tvs, tc, tn, con\ c\ \texttt{of}\ t\ \texttt{|}\ conbind, p, H \supset R)}$$

The relation $\mathsf{Hyp}(tvs, tc, tn, c, t, p, H)$ indicates that H is the case of the induction for the constructor c which takes an argument of type t—this may be a "base case" (if t is not recursive) or may include an induction hypothesis (if t is recursive). The encoding of Hyp is in a similar style to the encoding of Ind1.

This approach has the following features:

- Isabelle deduction is slow (compared to ML execution), and generating many rules in this way may slow the theorem prover unacceptably;
- as can be seen from the sample above, such Isabelle code is not especially perspicuous;
- the code which generates the rules is a formal object in Isabelle, and it is therefore simple to prove properties of it.

We leave open the question of which approach is best.

Labelled records need two rules:

- A rule stating that equality on labelled records is defined component-wise. This is like the injectivity rule for datatypes.
- A rule stating that a labelled record can always be decomposed into its components. This is like the induction rule for datatypes—except that, unlike datatypes, labelled record types cannot be recursive.

There is no need for a "no confusion" rule for labelled record types since they can only be formed in one way. Again, we can encode these rules via logic programs within Isabelle, or by calls to an "oracle" ML program.

7.3.2 Simplified ML

Suppose that we have a `case` expression of the form `case` *exp* `of` *match*, where *match* is pat_1 `=>` exp_1 `|` … pat_n `=>` exp_n. Suppose further that pattern pat_i contains the b-variables $b_1 : \tau_1$, …, $b_p : \tau_p$. We assert a rule of the form

$$\frac{}{decseq \vdash_C \forall b_1 \dots b_p{:}\, \tau_1 \dots \tau_p.\, \texttt{case}\ pat_i\ \texttt{of}\ match \approx exp_i}$$

(The pattern pat_i is regarded as an expression in this rule.) In Isabelle, this family of rules would be coded as:

$$decseq \vdash_C \mathsf{CaseProp}(i, match)$$

As usual, CaseProp is coded either within Isabelle or as an Isabelle oracle.

This is different to the rules for `case` in LAMBDA. There, `case` is "compiled" to $\varepsilon v{:}\,\tau.\, P$, where P is true if *exp* matches pat_i, and exp_i evaluates to v. Since the patterns are exhaustive, there is such a v, and since they are exclusive, there is just one such v. This approach is reasonable where such compilation is possible, but the equivalent rule in Isabelle is less elegant and directly applicable than our rules. The two approaches are logically equivalent.

The next group of rules concern `let` expressions.

Empty `let` introduction (`LetCong1`):

$$\frac{}{decseq \vdash_C \texttt{let nothing in } e \texttt{ end} \approx e}$$

Splitting `let` expressions (`LetCong2`):

$$\frac{}{fdecseq \vdash_C \texttt{let } bdec;\ \ bdecseq' \texttt{ in } e \texttt{ end} \approx \texttt{let } bdec \texttt{ in let } bdecseq' \texttt{ in } e \texttt{ end end}}$$

`let` introduction (`LetI`):

$$\frac{fdecseq;\ \ bdec\langle f_1 \ldots f_n / b_1 \ldots b_n\rangle \vdash_{C'} e[f_1 \ldots f_n / b_1 \ldots b_n] \approx e'}{fdecseq \vdash_C \texttt{let } bdec \texttt{ in } e \texttt{ end} \approx e'}$$

- $[v_i, \ldots, v_n] = \mathsf{Bound}(dec)$;
- For all $1 \leq i < j \leq n$, if $p_i = p_j$ then $v_i = v_j$;
- For each i, $p_i \notin e, e', dec$.

The first side-condition uses the semantic function Bound which gives a list of the variables bound by a declaration. Lists are written as in ML: $[a_1, \ldots, a_n]$. Sometimes set operations (intersection, union, and so on) are applied directly to lists: in these cases it is assumed that the list is silently converted to a set beforehand. This side-condition means that *dec* binds n variables.

The second side-condition formalizes the fact that the p_i are "at least as distinct" as the v_i they replace. When replacing identical b-variable names in two separate scopes, it does not matter if the replacing f-variables are distinct or not (since the scopes are separate). One might imagine that this is too weak, and that one can derive:

$$\frac{decseq;\ \texttt{local nothing in val}\ p_1\ \texttt{=}\ e_1\texttt{; val}\ p_2\ \texttt{=}\ e_2\ \texttt{end} \vdash_{C'} p_1 \approx e_1}{decseq \vdash_C \texttt{let local nothing in val}\ v\ \texttt{=}\ e_1\texttt{;}\ \ \texttt{val}\ v\ \texttt{=}\ e_2\ \texttt{end in}\ v\ \texttt{end} \approx e_1}$$

Informally, this says that from $e_1 \approx e_1$ one can derive $e_2 \approx e_1$. This is not possible, however, since $v[p_1/v, p_2/v]$ is p_2 and not p_1.

The requirement of the third side-condition that $p_i \notin dec$ is more restrictive than necessary: one need only ensure that the replacement of v_i by p_i does not capture free occurrences of p_i later in the declaration. This is a lot more complex to formulate, however, and so the simpler side-condition is used.

It might be supposed that side-conditions such as:

- $C' \vdash_{LOG} e[f_1 \ldots f_n/b_1 \ldots b_n] \approx e' \Longrightarrow o$ where $C' = C \oplus E$;
- $C \vdash_{STAT} dec\langle f_1 \ldots f_n/b_1 \ldots b_n\rangle \Longrightarrow E$;

would be needed in order to ensure that the proposition of the premiss is type correct: they are not necessary since they can be inferred from the fact that the `let` expression is type correct, and that no variables bound by *dec* appear in e'.

`let` **elimination** (`LetE`):

$$\frac{decseq \vdash_{C'} \texttt{let}\ dec\ \texttt{in}\ e\ \texttt{end} \approx e'}{decseq;\ \ dec\langle f_1 \ldots f_n/b_1 \ldots b_n\rangle \vdash_C e[f_1 \ldots f_n/b_1 \ldots b_n] \approx e'}$$

- $[v_i, \ldots, v_n] = \mathsf{Bound}(dec)$;
- For all $1 \leq i < j \leq n$, if $p_i = p_j$ then $v_i = v_j$;
- For each i, $p_i \notin e, e', dec$.

With the exception of the typing side-conditions, this is exactly `LetI` turned upside down. As with `LetI`, side-conditions to ensure the type correctness of the proposition of the premiss, such as:

- $C \vdash_{STAT} dec\langle f_1 \ldots f_n/b_1 \ldots b_n\rangle \Longrightarrow E$;
- $C' \vdash_{LOG} e \approx e' \Longrightarrow o$ where $C' = C \oplus E$;

are unnecessary.

Now we come to the rules which concern `local` declarations. First there are four rules with no logical content which perform the same function for `local` as `LetCong1` and `LetCong2` perform for `let`.

Empty `local` introduction (`LocI1`):

$$\frac{\mathit{decseq}\ \texttt{@}\ \mathit{decseq'} \vdash_C P}{\mathit{decseq};\ \texttt{local nothing in}\ \mathit{decseq'}\ \texttt{end} \vdash_C P}$$

Empty `local` elimination (`LocE1`):

$$\frac{\mathit{decseq};\ \texttt{local nothing in}\ \mathit{decseq'}\ \texttt{end} \vdash_C P}{\mathit{decseq}\ \texttt{@}\ \mathit{decseq'} \vdash_C P}$$

The rule `LocE1` is simply `LocI1` turned upside down. There are no side-conditions, since for both rules the left-hand-side of the premiss elaborates to the same environment as the left-hand-side of the consequent.

Unwinding `local` (`LocI2`):

$$\frac{\mathit{decseq};\ \texttt{local}\ \mathit{dec};\ \mathit{decseq'}\ \texttt{in}\ \mathit{decseq''}\ \texttt{end} \vdash_C P}{\mathit{decseq};\ \texttt{local}\ \mathit{dec}\ \texttt{in local}\ \mathit{decseq'}\ \texttt{in}\ \mathit{decseq''}\ \texttt{end end} \vdash_C P}$$

Winding up `local` (`LocE2`):

$$\frac{\mathit{decseq};\ \texttt{local}\ \mathit{dec}\ \texttt{in local}\ \mathit{decseq'}\ \texttt{in}\ \mathit{decseq''}\ \texttt{end end} \vdash_C P}{\mathit{decseq};\ \texttt{local}\ \mathit{dec};\ \mathit{decseq'}\ \texttt{in}\ \mathit{decseq''}\ \texttt{end} \vdash_C P}$$

Again, these rules are the inverse of each other, and again no side-conditions are necessary.

Now there are the rules which "do the work".

`local` introduction (`LocI`):

$$\frac{\mathit{decseq};\ \mathit{dec}\langle f_1 \ldots f_n / b_1 \ldots b_n \rangle\ \texttt{@}\ \mathit{decseq'}[f_1 \ldots f_n / b_1 \ldots b_n] \vdash_C P}{\mathit{decseq};\ \texttt{local}\ \mathit{dec}\ \texttt{in}\ \mathit{decseq'}\ \texttt{end} \vdash_{C'} P}$$

- $[v_i, \ldots, v_n] = \mathsf{Bound}(\mathit{dec})$;
- For all $1 \leq i < j \leq n$, if $p_i = p_j$ then $v_i = v_j$;
- For each i, $p_i \notin P, \mathit{dec}, \mathit{decseq'}$.

Side-conditions such as:

- $C' \vdash_{LOG} e \approx e' \Longrightarrow o$ where $C' = C \oplus E$;
- $C_{INIT} \vdash_{STAT} \mathit{decseq};\ \mathit{dec}\langle f_1 \ldots f_n / b_1 \ldots b_n \rangle\ \texttt{@}\ \mathit{decseq'}[f_1 \ldots f_n / b_1 \ldots b_n] \Longrightarrow E$;

are unnecessary, since none of the variables bound in P have changed. No variables used in P are hidden by the `local`, by the third side-condition. The substitution operation on *decseq′* is safe by the third side-condition, so no substitution of p_i can be captured.

`local` **elimination** (`LocE`):

$$\frac{\mathit{decseq};\ \texttt{local}\ \mathit{dec}\ \texttt{in}\ \mathit{decseq'}\ \texttt{end} \vdash_{C'} P}{\mathit{decseq};\ \ \mathit{dec}\langle f_1 \ldots f_n / b_1 \ldots b_n\rangle\ \ \texttt{@}\ \ \mathit{decseq'}[f_1 \ldots f_n / b_1 \ldots b_n] \vdash_C P}$$

- $[v_i, \ldots, v_n] = \mathsf{Bound}(\mathit{dec})$;
- For all $1 \leq i < j \leq n$, if $p_i = p_j$ then $v_i = v_j$;
- For each i, $p_i \notin P, \mathit{dec}, \mathit{decseq'}$.

Apart from the typing side-conditions, this rule is, again, the introduction rule upside down. Again, side-conditions such as:

- $C' \vdash_{LOG} e \approx e' \Longrightarrow o$ where $C' = C \oplus E$;
- $C_{INIT} \vdash_{STAT} \mathit{decseq};\ \texttt{local}\ \mathit{dec}\ \texttt{in}\ \mathit{decseq'}\ \texttt{end} \Longrightarrow E$;

are unnecessary. This rule is considerably less natural for backwards proof than the introduction rule, since one must decide which declarations are to move inside the `local` in advance.

Next, we have rules which deal with value declarations and recursive value declarations.

`val` **declarations** (`ValCong`):

$$\frac{\mathit{fdecseq} \vdash_{C'} e \approx_\tau e'}{\mathit{fdecseq};\ \ \texttt{val}\ f\ \ \texttt{=}\ \ e \vdash_C f \approx_\tau e'}$$

- $f \notin e'$;
- $C_{INIT} \vdash_{STAT} \mathit{fdecseq} \Longrightarrow E$;
- $C \oplus E = C'$.

Suppose the conclusion is type-correct. Then $f : \tau$, and so it must be the case that $e : \tau$, too. Hence the hypothesis is type-correct.

$$\frac{}{\mathit{fdecseq};\ \ \texttt{val}\ f\ \ \texttt{=}\ \ e \vdash_C f \approx e}$$

- $f \notin e$

There is another possible form of this rule:

`val rec` **declarations** (`ValRecCong`):

$$\frac{}{decseq;\ \texttt{val rec}\ p \texttt{=} e \vdash_C p \approx e}$$

This rule is just like the alternative form of `ValCong`, except that it does not need the side-condition, since any occurrence of p in e refers to this declaration, not a previous one.

A rule is needed for ML equality tests.

ML equality tests (`EqCong`):

$$\frac{}{decseq \vdash_C ((e\ \texttt{=}\ e') \approx\ \texttt{true})\ \approx\ (e \approx e')}$$

If e and e' are type correct, this rule states that ML equality on elements of equality type (`=`) is the same as the usual non-computable equality ($\approx$).

Finally, there are rules which allow us to reason about function abstractions. These rules have counterparts in the Higher-Order Logic part of the logic which allow reasoning about λ abstraction rather than `fn` abstraction.

ML β-conversion (`MLBetaCong`):

$$\frac{}{decseq \vdash_C \texttt{(fn}\ v\ \texttt{=>}\ e\texttt{)}\ e' \approx e[e'/v]}$$

ML η-conversion (`MLEtaCong`):

$$\frac{}{decseq \vdash_C \forall f\colon t \to t'.\,\texttt{fn}\ v\ \texttt{=> (}f\ v\texttt{)} \approx f}$$

7.3.3 Higher-Order Logic

Our logic is based in part on the logic of HOL [GM93]: the purely logical part of the logic (as opposed to the part concerned with ML) is described in this section.

Reflexivity (`Refl`):

$$\frac{}{decseq \vdash_C M \approx M}$$

There are no type side-conditions on this rule, although it is not well-typed unless M is well-typed. This is because it is only necessary that rules have the property that their premisses are type correct when their consequents are type correct. Since `Refl` has no premisses, this property is vacuously true.

β-conversion (`BetaConv`):

$$\overline{decseq \vdash_C (\lambda v\colon t.\, M)\ N \approx M[N/v]}$$

This rule illustrates one advantage of using the McKinna and Pollack binding strategy, rather than the more traditional Curry and Feys substitution: an "unsafe" (and very simple) substitution operation can be used, and still "free variables" (f-variables) of N can never be captured, because f-variables and b-variables are disjoint. Again, there are no type side-conditions.

Abstraction congruence (`AbsCong`):

$$\frac{decseq;\ \ \texttt{param}\ p\colon t \vdash_{C \oplus E} M[p/u] \approx M'[p/v]}{decseq \vdash_C (\lambda u\colon t.\, M) \approx (\lambda v\colon t.\, M')}$$

- $p \notin M$
- $p \notin M'$
- $C \vdash \texttt{param}\ p\colon t \Longrightarrow E$

Substitution (`Subst`):

$$\frac{decseq \vdash_C M \approx M' \quad decseq \vdash_C P[M/p]}{decseq \vdash_C P[M'/p]}$$

- $C \vdash_{LOG} M \approx M' \Longrightarrow o$;
- $P[M/p]$ is well-formed;
- $P[M'/p]$ is well-formed.

Naturally, this rule is horrid. In order to use it in backwards proof, it is necessary to first "unsubstitute" a term for a (new) parameter in the consequent before the rule can be applied. This cannot be done without providing some guidance to the prover—namely, which term is being substituted. That this rule is so unpleasant is the result of two factors:

1. the only substitution operations available can only substitute for an f-variable or a b-variable;
2. substitution is not a primitive, and so putting a term into the form $P[M/p]$ cannot be a trivial matter, but actually requires some proof effort.

The well-formedness side-conditions of this rule require some justification. They are needed because, in a call-by-value language such as ML, the usual substitution rule is not safe. Consider the following which can be derived by using `MLBetaConv` (assuming `n` $\notin$ `e2`), and *decseq* contains the declaration `datatype Nat = Z | S of Nat` (type names are omitted for brevity):

decseq $\vdash_C$ `(fn n => fn e1 => fn e2 =>`
```
            case n of
            Z     => e1
            | S m => e2[m/p])
```
$b\ \ e_1\ \ e_2$

$\approx$

`case` b `of Z =>` e_1 `| S m =>` e_2`[m/`p`]`

Now consider the judgement:

decseq $\vdash_C$ `let`
```
        val rec factorial = fn n =>
            case n of
            Z     => S Z
            | S m => (S m) * (factorial m)
    in
        factorial Z
    end
```
$\approx$ `S Z`

Using the rule `Subst` without the well-formedness conditions gives:

decseq $\vdash_C$ `let`
```
        val rec factorial = fn n =>
            (fn b => fn e1 => fn e2 =>
                case b of
                Z     => e1
                | S m => e2)
            n (S Z) ((S m) * factorial m)
    in
        factorial Z
    end
```
$\approx$ `S Z`

This does not terminate for any argument (since it always evaluates e_2). The problem is in the use of the rule `MLBetaConv`, where a non-strict construct, `case`, is equated with a strict construct, function application. Usually this simply changes the order of evaluation, but in the case that the term in which the substitution takes place contains declarations of recursive functions (via `let` expressions), it can lead to non-terminating functions. A check that the terms after substitution are still well-formed (and thus that any Simplified ML declarations still terminate) is then necessary. I believe that this is the simplest side-condition which ensures soundness and still retains some form of referential transparency.

The order of evaluation in Standard ML is given by the Definition. It is not trivial that changing the order of evaluation via `Subst` does not change the value of a term: some version of the Church-Rosser Theorem is needed to ensure that this is sound. The Church-Rosser Theorem does not hold for Standard ML because both exceptions and state can cause differences in the value of a term under different orders of evaluation. Although I do not prove it, there is every reason to believe that the theorem holds for Simplified ML, since Simplified ML bears more than a passing resemblance to a λ-calulus such as System F, which is confluent.

As it stands, this rule is not sound for another reason, however. Suppose *decseq* is:

```
val f = fn v => (); val f' = fn v => v
```

The function `f` has type scheme $\sigma = \forall \alpha.\alpha \to \texttt{unit}$ (in an appropriate context), and `f'` has type scheme $\sigma' = \forall \alpha.\alpha \to \alpha$. The type schemes σ and σ' have a common instance $\texttt{unit} \to \texttt{unit}$, and for this type they are equal, since there is only one total function $\texttt{unit} \to \texttt{unit}$. Hence $decseq \vdash_C \texttt{f} \approx \texttt{f'}$ is type correct and valid, for the appropriate context C.

Now suppose P is $p\ \texttt{1} \approx p\ \texttt{2}$, M is `f` and M' is `f'`. $P[M/p]$ is $\texttt{f 1} \approx \texttt{f 2}$, which is $\texttt{()} \approx \texttt{()}$. Hence $decseq \vdash_C P[M/p]$ is type correct and true. On the other hand, $P[M'/p]$ is $\texttt{f' 1} \approx \texttt{f' 2}$, which is type correct, despite the fact that its elaboration is different to that of $P[M/p]$, and is false.

There is a simple modification which prevents this—replace the first side-condition by:

- $C \vdash_{LOG} M \Longrightarrow \tau$;
- τ principal for M in C;
- $C \vdash_{LOG} M' \Longrightarrow \tau$;

- τ principal for M' in C.

This means that M has exactly the same types that M' has. I believe that this solution may also require the principality side-conditions described in Section 7.2. That is to say, the context C must be of the form $C_{INIT} \oplus E$, where E is principal for *decseq* in C_{INIT}.

Conceivably, there may be less restrictive ways to achieve soundness, but we adopt this one.

Discharging an assumption (`ImpI`):

$$\frac{\begin{array}{c}[\mathit{decseq} \vdash_C P] \\ \vdots \\ \mathit{decseq} \vdash_C Q\end{array}}{\mathit{decseq} \vdash_C P \supset Q}$$

Modus Ponens (`ImpE`):

$$\frac{\mathit{decseq} \vdash_C P \quad \mathit{decseq} \vdash_C P \supset Q}{\mathit{decseq} \vdash_C Q}$$

- $C \vdash_{LOG} P \Longrightarrow o$

In addition to these rules, there are four other axioms, before which, strictly, we must give definitions for some constants used in the axioms:

$$\begin{array}{lcl}
\mathit{decseq} \vdash_C \mathsf{T} & \approx & ((\lambda x{:}\, o.\, x) \approx (\lambda x{:}\, o.\, x)) \\
\mathit{decseq} \vdash_C \forall & \approx & \lambda P{:}\, \alpha \to o.\, P \approx (\lambda x{:}\, \alpha.\, \mathsf{T}) \\
\mathit{decseq} \vdash_C \exists & \approx & \lambda P{:}\, \alpha \to o.\, P(\varepsilon P) \\
\mathit{decseq} \vdash_C \mathsf{F} & \approx & \forall P{:}\, o.\, P \\
\mathit{decseq} \vdash_C \neg & \approx & \lambda P{:}\, o.\, P \supset \mathsf{F} \\
\mathit{decseq} \vdash_C \wedge & \approx & \lambda P, Q{:}\, o.\, \forall R{:}\, o.\, (P \supset Q \supset R) \supset R \\
\mathit{decseq} \vdash_C \vee & \approx & \lambda P, Q{:}\, o.\, \forall R{:}\, o.\, (P \supset R) \supset (Q \supset R) \supset R
\end{array}$$

Now it is possible to state the four remaining axioms:

The law of the excluded middle (`OInd`):

$$\frac{}{\mathit{decseq} \vdash_C \forall b{:}\, o.\, (b \approx \mathsf{T}) \vee (b \approx \mathsf{F})}$$

We follow the HOL formulation for the η rule, making it an axiom, rather than a rule with premisses.

η-conversion (`EtaConv`):

$$\frac{}{\mathit{decseq} \vdash_C \forall f{:}\, t \twoheadrightarrow t'.\, \lambda v{:}\, t.\, (f v) \approx f}$$

Another feature of McKinna and Pollack's style of binding is that v *cannot* appear free in f: v is a b-variable, and f must be b-closed.

Implication is antisymmetric (`ImpAntiSym`):

$$\frac{}{decseq \vdash_C \forall P, Q\colon o.\, (P \supset Q) \supset (Q \supset P) \supset (P \approx Q)}$$

Selection (`Select`):

$$\frac{}{decseq \vdash_C \forall P\colon \alpha \to o.\, \forall x\colon \alpha.\, Px \supset P(\varepsilon P)}$$

7.3.4 Structural rules

The reader will have noticed that some of the rules given for reasoning about Simplified ML are not powerful enough on their own to make interactive proof a serious possibility. These rules have been chosen to simplify soundness arguments. It is intended that the rules for Simplified ML should be combined with the following structural rules in automatic proof tactics to perform useful steps.

Weakening the context (`Weak`):

$$\frac{decseq;\quad dec \vdash_{C'} P}{decseq \vdash_C P}$$

- $\mathsf{Bound}(dec) \notin P$
- $C \vdash_{STAT} dec \Longrightarrow E$
- $C' = C \oplus E$

There is no need to check that P elaborates correctly in the premiss, since it does not refer to any variable in $\mathsf{Bound}(dec)$ so its elaboration remains unchanged.

Strengthening the context (`Stren`):

$$\frac{decseq \vdash_{C'} P}{decseq;\quad dec \vdash_C P}$$

- $\mathsf{Bound}(dec) \notin P$
- $C_{INIT} \vdash_{STAT} decseq \Longrightarrow E$
- $C' = C_{INIT} \oplus E$

Again, there is no need to check the elaboration of P, since C is identical to C' for everything referenced in P.

Permuting the context (`Exch`):

$$\frac{decseq; \;\; dec; \;\; dec' \vdash_C P}{decseq; \;\; dec'; \;\; dec \vdash_C P}$$

- $\mathsf{Bound}(dec) \notin dec'$
- $\mathsf{Bound}(dec') \notin dec$
- $\mathsf{Bound}(dec) \cap \mathsf{Bound}(dec') = \emptyset$

No type side-conditions should be necessary, since changing the order of *dec* and *dec'* affects neither which bindings are in scope (the third side-condition), nor the meaning of those bindings (the first two side-conditions).

7.4 Models of the Logic

The model used for the logic is a simple set-theoretic one: an extension of the model for HOL given by Pitts [GM93].

In short, types are represented as follows:

- datatypes are represented by least fixed points of their constructors (this is possible because of the restrictions we have placed on them to ensure that they are inductive);
- compound monotypes (records, functions) are non-empty sets of values, constructed in the obvious way;
- polymorphic types are represented by functions from types to types, as are parametrized datatypes.

Since we have in ML a strict separation of types and type schemes, we do not run into the problem that polymorphism is not set-theoretic [Rey83].

It is necessary to define the representations of terms and contexts together. A context is a mapping from variables to values (representations of terms). Terms may need to refer to the context in which they are defined. The valuation function is defined by induction on the structure of terms, and its details are left to the interested reader. Note that, whenever `param` declarations appear, the context becomes a function from values (representing the free `param`) to ordinary contexts.

Part III

Data Reification

Chapter 8

Data reification

Data reification is a well-known formal method used, for example, in VDM [Jon86] and going back to Hoare's 1972 paper [Hoa72]. In this chapter, I apply data reification to Simplified ML.

ML Modules provide an ideal method of structuring programs for reification: I define what it means for one Simplified ML structure to reify another. ML has several features which must be accounted for:

- polymorphic types, such as $\alpha \to \alpha$;
- higher-order functions, such as $(\texttt{int} \to \texttt{bool}) \to \texttt{int}$;
- parametrised datatypes, such as `int list`.

These features mean that a reification cannot simply be a set of retrieve functions, as in VDM. We develop *logical structure relations*, based on logical relations [Sta85], which enable us to give a family of relations, one for every Simplified ML type. A reification is then given by a logical structure relation with certain properties.

One surprising conjecture is that, for a large class of reifications, reification is transitive. As Tennent [Ten94] notes, transitivity does not in general hold when ordinary logical relations are used. All though we do not prove this conjecture, we provide some evidence that it holds.

In order to define logical structure relations, we must examine both which values of types over a structure are visible outside that structure, and which values represent the same (abstract) value. This leads us to define the notion of *abstract equivalence* for every type over a structure, and, from that, *well-behavedness*. Well-behavedness is a generalization of VDM's data invariants.

8.1 A brief introduction to data reification

In this section we outline the method of data reification, as used in VDM. For a more detailed account, the reader should consult Jones' VDM book [Jon86]. We describe reification for first order types and operations on those types (as in VDM). However, since Simplified ML is a total, purely functional language without exceptions, we will not discuss non-termination or error values.

We begin by discussing which values of a type represent abstract objects, and show how *data invariants* separate those values from the meaningless "junk" values of a type.

We introduce the notions of *abstract* and *concrete* implementations. We then introduce the use of *retrieve functions* to give the relationship between an abstract implementation and a concrete implementation which reifies it.

The discussion is not totally informal, and we will use an ML-like syntax (as opposed to VDM syntax). It is deliberate that this suggests how reification will work for ML.

8.1.1 Data invariants

Data reification relates elements of a "concrete" type to those of an "abstract" type. But which elements do we wish to relate? Some values of a type might be considered meaningless junk.

Suppose we represent finite sets by ordered lists without repetition. An unordered list does not represent any set. Furthermore, operations on such values can lead to unpredictable results.

For example, suppose the membership function looks down the list until it either finds the value it is looking for, or finds an element larger than the one it is looking for. Then we have the following:

$$\texttt{member (1, [3, 1, 2])} \neq \texttt{member (1, remove (3, [3,1,2]))}$$

To separate the "useful" values, every type must have a *data invariant*. The data invariant should be true for exactly the useful values.

We then require that the data invariant holds for values of the type which are made available to the outside world (in ML, this is those values which are visible in the signature of a structure). We also require that the operations available to the outside world preserve data invariants. That is, if they are passed an argument for which the relevant data invariant is true, they must return a result for which the invariant is true. VDM requires that *all* operations preserve invariants, but we can relax that restriction: if the outside world cannot see a function, it can do whatever it pleases behind closed doors.

This means that code which uses values of this type can never have access to junk values: it can only use values for which the invariant is true, and the operations it can use on these values cannot falsify the invariant.

8.1.2 Reification for first order types

Data reification [Jon86] is a method for showing that a "concrete" datatype ($C.t$, say) represents an "abstract" one ($A.t$). Here "concrete" means roughly "more like executable code", so that the most concrete datatypes are programs; "abstract" means "more like a specification". In order to show that $C.t$ is a reification of $A.t$ we must give a "retrieve function" from the concrete to the abstract (in fact, from the part of the concrete datatype for which the data invariant is true to the part of the abstract datatype for which the data invariant is true).

Furthermore, we must ensure that we are representing everything in the abstract world: that is, for every abstract value there is at least one concrete value which is mapped to it by the retrieve function. This amounts to showing that the retrieve function is surjective.

What of operations on these datatypes? In order for a concrete function $C.f : C.t1 \to C.t2$ to be a reification of an abstract function $A.f : A.t1 \to A.t2$ we must show that, starting from some concrete value $x : C.t1$ we obtain the same abstract value by applying the retrieve function and then the abstract function $A.f$ as we do by applying the concrete function $C.f$ and then mapping back to the abstract. This is summarized by the commutativity of the following diagram.

$$\begin{tikzcd} C.t1 \arrow[r, "C.f"] \arrow[d, "\mathsf{retr}_{t1}"'] & C.t2 \arrow[d, "\mathsf{retr}_{t2}"] \\ A.t1 \arrow[r, "A.f"] & A.t2 \end{tikzcd}$$

In fact, where functions are possibly non-terminating, rather than requiring that $\mathsf{retr}_{t2} \circ C.f = A.f \circ \mathsf{retr}_{t1}$, VDM requires that $\mathsf{retr}_{t2} \circ C.f$ is at least as defined as $A.f \circ \mathsf{retr}_{t1}$: that is, whenever $A.f \circ \mathsf{retr}_{t1}$ terminates, $\mathsf{retr}_{t2} \circ C.f$ also terminates and the two functions return the same result . Since we are only concerned with terminating functions, we omit further discussion of this.

In order to show a reification, then, there are several steps.

1. For each of the types t_i in the reification there is a data invariant, WB_{t_i}.

2. Each operation in the reification $f : t1 \to t2$ (in both A and C) maps invariant values to invariant values.

3. For each type t we are reifying, we must give a retrieve function $\mathsf{retr}_t : C.t \twoheadrightarrow A.t$.

4. Each retr_t, when its domain is restricted to the part of $C.t$ for which the data invariant holds, is a surjective function to the part of $A.t$ for which the data invariant holds.

5. The diagram above commutes, when restricted to the invariant parts of types.

8.1.3 ML

Essentially, in data reification, we are showing that some group of types and values can be represented by some corresponding types and values. But in ML a "group of types and values" is a structure; and a "corresponding group of types and values" is another structure with the same signature as the first. Thus, we formalize reification in Simplified ML as a relationship between structures having the same signature.

This seems the natural approach to formalizing reification in ML. Not only that, but we also believe that this approach gives us insight into the method of reification in general: it effectively separates the abstract and the concrete, and it shows which parts of a program are to be reified (that is, a signature shows the interface of a structure).

8.2 Types with holes

Suppose we have two structures S_1 : SIG and S_2 : SIG, where the signature SIG contains the type α `tree` and the value `emptyTree` : α `tree` (and the types `int`, `list`, and `bool` are pervasive, as usual). When we try to show that S_2 is a reification of S_1, we will certainly need to show that S_1.`emptyTree` is related to S_2.`emptyTree`. The type of such a relation is $\alpha\ S_1.\mathtt{tree} \twoheadrightarrow \alpha\ S_2.\mathtt{tree} \twoheadrightarrow o$. Here are some other types we might wish to relate (assuming S_1 is in the scope of S_2).

Type in S_1	Type in S_2	Description
`int` S_1.`tree`	`int` S_2.`tree`	integer trees
α S_1.`tree list`	α S_2.`tree list`	lists of trees
α S_1.`tree` S_1.`tree`	α S_1.`tree` S_2.`tree`	one kind of tree
$\alpha \to \alpha$ S_1.`tree` $\to$ `bool`	$\alpha \to \alpha$ S_2.`tree` $\to$ `bool`	set membership

It will be seen that there are several kinds of type which need considering:

$$
\begin{array}{lcl}
\textit{holety} & ::= & \textit{tyvar} \\
 & & \{\langle \textit{holetyrow} \rangle\} \\
 & & \textit{holetyseq holetycon} \\
 & & \textit{holety}_1 \ \rightarrow \ \textit{holety}_2 \\
 & & (\textit{holety}) \\
 & & \\
\textit{holetyrow} & ::= & \textit{lab} \ : \ \textit{holety} \ \langle \ , \ \textit{holetyrow} \rangle \\
\textit{holetycon} & ::= & \textit{longtycon}_{\textit{tyname}} \\
 & & [].\textit{longtycon}
\end{array}
$$

As in the Definition, an Xseq is either a single X, or a series of Xs separated by commas and terminated by brackets.

Figure 8.1: The grammar of types with holes

- polymorphic types;
- higher-order function types;
- parametrised datatypes;
- type constructors which are the same in the S_1 column of the table and the S_2 column, such as S_1.`tree` in the third example and `bool` in the last;
- type constructors which differ in the S_1 and S_2 columns, such as S_1.`tree` and S_2.`tree` in the first example.

In order to capture the difference between the last two cases, we introduce *types with holes*, where type constructors may appear with a "hole" in place of the outermost structure name: $[].\textit{longtycon}$. A type constructor with a hole instead of a structure name stands for "the type constructor with this identifier in each of the structures to be related". The syntax of types with holes is given in Figure 8.1. We are still working in a world with explicit type names: type constructors with holes must have their type names filled in when the hole is filled in with a structure name.

Ordinary Simplified ML types form a subset of types with holes: every type is also a type with holes which has no holes. A type with holes over a signature SIG is a type with holes in which every type constructor which appears with a hole, $[].t$, is specified in SIG. Given a type with holes, τ, over a signature SIG, and a structure S : SIG, we can "fill in the holes" in τ to give a type $\tau[S]$ as follows:

$$
\begin{array}{lcl}
\alpha[S] & = & \alpha \\
\{lab_1 : \tau_1, \ldots, lab_n : \tau_n\}[S] & = & \{lab_1 : \tau_1[S], \ldots, lab_n : \tau_n[S]\} \\
(\tau_1 \ldots \tau_n\ t_{tn})[S] & = & \tau_1[S] \ldots \tau_n[S]\ t_{tn} \\
(\tau_1 \ldots \tau_n\ [].t)[S] & = & \tau_1[S] \ldots \tau_n[S]\ S.t_{tn} \\
 & & \text{where } tn \text{ is the name of the type } t \text{ defined in } S \\
(\tau' \to \tau'')[S] & = & \tau'[S] \to \tau''[S]
\end{array}
$$

We can now return to our table of types, with the types with holes written down.

Type in S_1	Type in S_2	Type with holes
`int` S_1`.tree`	`int` S_2`.tree`	`int` $[]$`.tree`
α S_1`.tree list`	α S_2`.tree list`	α $[]$`.tree list`
α S_1`.tree` S_1`.tree`	α S_1`.tree` S_2`.tree`	α S_1`.tree` $[]$`.tree`
$\alpha \to \alpha$ S_1`.tree` $\to$ `bool`	$\alpha \to \alpha$ S_2`.tree` $\to$ `bool`	$\alpha \to \alpha$ $[]$`.tree` $\to$ `bool`

In effect, the types appearing in a signature are also types with holes. We can see this by the following reasoning. Suppose we have a signature SIG. There could be any number of structures with this signature, each of which will have a different instantiation of the types specified in SIG. So each type t in SIG represents an infinite family of types $[].t$. The types of values specified in SIG may include references to the types specified in SIG: naturally (because of the scope of the signature declaration) these references will not have structure names (or type names). For example, consider the following simple signature:

```
signature SIG = sig
  type t
  val x:  t
end;
```

Now suppose we have structures $S1$: SIG and $S2$: SIG. The structure $S1$ includes a type $S1.t$ and a value $S1.x : S1.t$; $S2$ contains a type $S2.t$ and a value $S2.x : S2.t$. If we now consider the signature to specify a type $[].t$ and a value $[].x : [].t$, we can read this as: $S1$ contains a type $([].t)[S1]$ and a value $([].x)[S1] : ([].t)[S1]$, and similarly for $S2$. More generally, if a signature SIG declares a value $x : \tau$ (where x and τ have holes as appropriate), then a structure S : SIG will contain a value $x[S] : \tau[S]$. Once we are outside the scope of the signature declaration, we interpret a declaration $v : \tau$ as a declaration of $[].v$ where τ is considered a type with holes. We can only fill in a particular structure name and type name when we have a particular structure matching this signature, (and they are left out in the signature declaration).

The observant reader will have noticed that we have used "expressions with holes" in the above (the value $[].x$ for example). These are defined in an analogous way to types with holes: they have the same grammar as ordinary expressions, but where a *longcon* can appear, the "holey" form $[].\mathit{longcon}$ can also appear. Expressions with holes over a signature, and hole-filling for expressions with holes are likewise defined. Type-checking is informally understood as follows: given an expression e with holes over a signature SIG, assume that there is an "extra" structure $[] : \mathrm{SIG}$ in scope, and continue type-checking as normal.

8.3 Data reification for all ML types

In this section, I introduce the ideas which I will use to define reification in Simplified ML in the following sections. I answer the question: "how do we give a retrieve relation at a type with holes τ?" The answer to this question will allow us to give a formal definition of reification in section 8.4. In the following, the relation R_τ is the relation with which we represent reification at a type with holes τ (our equivalent of a retrieve function for τ).

8.3.1 Labelled record types

This is the simplest case. When τ is $\{lab_1 : \tau_1; \ldots; lab_r : \tau_r\}$, then reification is defined point-wise: $a : \tau[A]$ is related to $c : \tau[C]$ when every component $\#lab_i(a)$ is related to $\#lab_i(c)$.

8.3.2 Function types and well-behavedness

We wish to define a retrieve relation for a type with holes $\tau' \to \tau''$, given the relation for τ' and τ''. Recall the way that reification is defined in VDM for operations on first order types:

$$
\begin{tikzcd}
C.t1 \arrow[r, "C.f"] \arrow[d, "\mathsf{retr}_{t1}"'] & C.t2 \arrow[d, "\mathsf{retr}_{t2}"] \\
A.t1 \arrow[r, "A.f"] & A.t2
\end{tikzcd}
$$

That is to say, $C.f$ and $A.f$ are related if they map related arguments to related results. However, we are only concerned with arguments which are "well behaved"—the exact meaning of well-behaved will be defined later, but well-behaved values will certainly satisfy our notion of the datatype invariant.

First, let us examine what happens if we do not weed out junk values, and quantify over *all* possible arguments:

$$
\begin{array}{l}
a\ R_{\tau' \to \tau''}\ c \\
\iff \\
\forall a'\!:\tau'[A].\, \forall c'\!:\tau'[C]. \\
\quad a'\ R_{\tau'}\ c' \\
\quad \supset (aa')\ R_{\tau''}\ (cc')
\end{array}
$$

Families of relations such as R_τ are called *logical relations* by Statman [Sta85]. They are used by Tennent [Ten94] to define reification for higher-order function types in Algol. However, as Tennent notes, using logical relations means that transitivity does not hold. That is: if A is reified by B (via a logical relation R), and B is reified by C (via a logical relation R'), it is not necessarily the case that A is reified by C. This failure is disastrous since it means that we cannot proceed by step-wise refinement.

Transitivity fails because we cannot in general give a logical relation $R \circ R'$ such that $(R \circ R')_\tau$ includes $R_\tau \circ R'_\tau$. The following is a simple counter-example which shows that $(R \circ R')_\tau$ is not in general equal to $R_\tau \circ R'_\tau$.

Counter-example: consider the case at type $\tau = \tau' \to \tau''$ when $(R \circ R')_{\tau'} = R_{\tau'} \circ R'_{\tau'}$ and $(R \circ R')_{\tau''} = R_{\tau''} \circ R'_{\tau''}$, and both $R'_{\tau'}$ and $R'_{\tau''}$ are the empty relation.

Then $R'_{\tau' \to \tau''}$ is the maximal (always true) relation, and $R_{\tau' \to \tau''} \circ R'_{\tau' \to \tau''}$ is $\lambda x.\, \exists y.\, x\ R_{\tau' \to \tau''}\ y$. However, $(R \circ R')_{\tau'}$ is the empty relation, as is $(R \circ R')_{\tau''}$. This means that $(R \circ R')_{\tau' \to \tau''}$ is the always true relation.

Hence, $(R \circ R')_{\tau' \to \tau''} \neq R_{\tau' \to \tau''} \circ R'_{\tau' \to \tau''}$.

Although this counter-example demonstrates that there are situations in which transitivity fails, the empty relation certainly is not a reification. The next counter-example involves relations which could plausibly be thought of as reifications.

Counter-example: suppose SIG is a signature containing (nullary) type constructors t, t' and t''. Suppose further that A, B, and C are structures having signature SIG. We will give logical relations R and R' such that there is an $x : (A.t \to A.t') \to A.t''$ and a $z : (C.t \to C.t') \to C.t''$ with $x\ (R_{(t \to t') \to t''} \circ R'_{(t \to t') \to t''})\ z$ but not $x\ (R \circ R')_{(t \to t') \to t''}\ z$.

(We have not yet defined composition of logical relations. For the present, we need only know that, for any nullary type constructor s, $(R \circ R')_s = R_s \circ R'_s$).

Let A contain the following datatype declarations.

$\texttt{datatype}\ t\ \ = a$

$\texttt{datatype}\ t'\ = a'$

$\texttt{datatype}\ t'' = a''_1$
$\quad\quad\quad\quad\quad\ \mid a''_2$

There is only one (total) function p having type $A.t \to A.t'$, and only two functions having type $(A.t \to A.t') \to A.t''$: one mapping p to a''_1 and one mapping it to a''_2.

Let B contain the following datatype declarations.

$\texttt{datatype}\ t\ \ = b$

$\texttt{datatype}\ t'\ = b'_1$
$\quad\quad\quad\quad\quad\ \mid b'_2$

$\texttt{datatype}\ t'' = b''_1$
$\quad\quad\quad\quad\quad\ \mid b''_2$

Let C contain the following datatype declarations.

$\texttt{datatype}\ t\ \ = c_1$
$\quad\quad\quad\quad\quad\ \mid c_2$

$\texttt{datatype}\ t'\ = c'_1$
$\quad\quad\quad\quad\quad\ \mid c'_2$

$\texttt{datatype}\ t'' = c''_1$
$\quad\quad\quad\quad\quad\ \mid c''_2$

Let R be given by the following relations (using set-theoretic notation):

$$\begin{array}{rcl} R_t &=& \{(a, b)\} \\ R_{t'} &=& \{(a', b'_1), (a', b'_2)\} \\ R_{t''} &=& \{(a''_1, b''_1), (a''_2, b''_1)\} \end{array}$$

Each of these is a total surjective function from right to left, and so it represents what we think of as a reification.

Let R' be given by the following relations:

$$\begin{array}{rcl} R'_t & = & \{(b, c_1), (b, c_2)\} \\ R'_{t'} & = & \{(b'_1, c'_1), (b'_2, c'_2)\} \\ R'_{t''} & = & \{(b''_1, c''_1), (b''_2, c''_2)\} \end{array}$$

Again, each of these is a total surjective function from right to left.

The relations R_t, $R_{t'}$ and R'_t are all maximal relations, so $R_t \circ R'_t$ and $R_{t \to t'}$ are also maximal relations. Also, $R_{t'} \circ R'_{t'}$ is maximal, but $R'_{t'}$ is not.

This means that $(R \circ R')_{t \to t'}$ is maximal, too. However, consider the following function $r : C.t \to C.t'$.

fun r c_1 = c'_1
| $\quad r$ c_2 = c'_2

There is no value $q : B.t \to B.t'$ with $q\ {R'}_{t \to t'}\ r$. Suppose there were. Since $b\ {R'}_t\ c_1$, this would mean that $(qb)\ {R'}_{t'}\ (rc_1)$, which would mean that qb would have to be b'_1. But also, $b\ {R'}_t\ c_2$, which would mean that qb would have to be b'_2, which is a contradiction. Thus, by the definition of composition, it is not the case that $p\ (R_{t \to t'} \circ R'_{t \to t'})\ r$.

We have demonstrated a p and r such that $p\ (R \circ R')_{t \to t'}\ r$, but not $p\ (R_{t \to t'} \circ R'_{t \to t'})\ r$. We now give $x : (A.t \to A.t') \to A.t''$ and $z : (C.t \to C.t') \to C.t''$ with $x\ (R_{(t \to t') \to t''} \circ R'_{(t \to t') \to t''})\ z$ but not $x\ (R \circ R')_{(t \to t') \to t''}\ z$. Let x be the following.

fun x p = a''_1

Let z be the following.

fun z f =
$\quad$ if f c_1 = c'_1 andalso f c_2 = c'_2
$\quad$ then c''_2
$\quad$ else c''_1

The function z returns c''_1 unless it is passed r, when it returns c''_2.

It is not the case that $x\ (R \circ R')_{(t \to t') \to t''}\ z$, since $p\ (R \circ R')_{t \to t'}\ r$, but not $(xp)\ (R \circ R')_{t''}\ (zr)$.

We show $x\ (R_{(t \to t') \to t''} \circ R'_{(t \to t') \to t''})\ z$ by giving a $y : (t \to t') \to t''$ with $x\ R_{(t \to t') \to t''}\ y$, and also $y\ {R'}_{(t \to t') \to t''}\ z$. The following is such a y.

`fun` $y\ q$ `=` b''_1

To see this, consider an arbitrary $f : A.t \to A.t'$ and $g : B.t \to B.t'$. (Since $R_{t\to t'}$ is maximal, $f\ R_{t\to t'}\ g$ whatever f and g are). By the definition of x, $xf = a''_1$, and similarly $yg = b''_1$, so $(xf)\ R_{t''}\ (yg)$.

Now consider a $g : B.t \to B.t'$ and an $h : C.t \to C.t'$ with $g\ R'_{t\to t'}\ h$. Since g and h are related $h \neq r$, and so $zh = c''_1$. Also, $yg = b''_1$, and so $(yg)\ R'_{t''}\ (zh)$.

This counterexample hinges on the functions r and z.

Intuitively, because they are related to the same value in $A.t'$, we think of $B.b'_1$ and $B.b'_2$ as representing the same abstract value. We also think of $C.c'_1$ and $C.c'_2$ as representing the same abstract value, since they are related to values which represent the same abstract value. So $B.b'_1$ represents the same abstract value as something ($B.b'_2$) which is related to $C.c'_2$, but $B.b'_1$ is not related to $C.c'_2$. There are two reasons why r is not related to anything.

- We have not formalized the relation "represents the same abstract value as".

- We might hope that given two related values, every pair of values which represent the same abstract values as them are also related. This is not the case, and indeed by requiring $R'_{t'}$ to be a total surjective function, we rule out this possibility. (More mathematically, "represents the same abstract value as" on $B.t'$ should be $\mathsf{lper}(R'_{t'})$ and on $C.t'$ should be $\mathsf{rper}(R'_{t'})$, and $R'_{t'}$ should be many-step closed).

Now consider the function z. It maps values in $C.t \to C.t'$ which represent the same abstract value to values in $C.t''$ which *do not* represent the same abstract value. For example, suppose that this example was about sets. One might, for example, have two functions which added a and b to a set: one by adding a and then b, the other by adding b then a to the set. These functions would clearly represent the same abstract value, and should be indistinguishable. The function z, however, might give different results for two such values. This is obviously undesirable, and we should restrict our attention to functions which are not badly behaved in this way.

When defining $R'_{\tau'\to\tau''}$, we should only quantify over functions which are well-behaved. We will define the well-behavedness predicate WB_τ for every type in Section 8.4. The definition of $R_{\tau'\to\tau''}$ then becomes the following.

$a \; R_{\tau' \to \tau''} \; c$
$\iff$
$\forall a' : \tau'[A] \mid \mathrm{WB}. \forall c' : \tau'[C] \mid \mathrm{WB}.$
$\quad a' \; R_{\tau'} \; c'$
$\quad \supset (aa') \; R_{\tau''} \; (cc')$

What exactly do we mean by "well behaved" functions? Simply that they map arguments we consider equivalent to results we consider equivalent. The meaning of "equivalent" depends on the particular types under consideration, of course. Suppose Eq_τ is the equivalence for type τ, and the well-behavedness predicate at τ is WB_τ. When τ is the function type $\tau_1 \to \tau_2$, $\mathrm{WB}_\tau(f)$ is equivalent to:

$$\forall x, y : \tau_1 .\; x \; \mathrm{Eq}_{\tau_1} \; y \supset fx \; \mathrm{Eq}_{\tau_2} \; fy$$

This is very similar to our original definition of logical relations. We are left with three linked concepts.

- Well behavedness predicates.
- Datatype invariants.
- An "abstract" notion of equivalence.

We can use the same relation for all of these concepts. We will require that every datatype has its own abstract equivalence, Eq. These relations will be partial equivalences (that is, symmetric and transitive but not necessarily reflexive). Given the abstract equivalence for datatypes, the abstract equivalence for all other types will be defined in a similar (but not identical) way to logical structure relations. Datatype invariants and well-behavedness predicates will be identified, and defined as $\mathrm{WB}_\tau(x) = x \; \mathrm{Eq}_\tau \; x$. Values which are not abstractly equivalent to anything (and in particular themselves) will not be well-behaved. Note that since Eq_τ always relates values of the same type, τ is an ordinary type, rather than a type with holes.

One consequence of the definition of logical structure relations for function types is that, under certain circumstances, transitivity does hold. This is proved in Section 9.3. This result means that we can, after all, develop programs by step-wise refinement.

8.3.3 Parametrised datatypes

There are two cases:

- The type with holes τ is $(\tau_1, \ldots, \tau_n)[].t$; that is to say, the type constructor t has a hole.

- The type with holes τ is $(\tau_1, \ldots, \tau_n)t$; that is to say, the type constructor t does not have a hole. Notice that τ may still have holes within it, since the parameter types τ_i may have holes.

8.3.3.1 Type constructors occurring with a hole

We first consider the case that t has a hole: the second case will make use of similar techniques to those used in the first.

As an example, we consider relating sets represented as characteristic functions (structure A) and sets represented as lists without repetition of elements (structure B). Supposing τ_1 is a type with holes, what should the relation between $(\tau_1[A])A.\mathtt{set}$ and $(\tau_1[B])B.\mathtt{set}$ be?

The simplest case is when τ_1 contains no holes, and so $\tau_1[A]$ and $\tau_1[B]$ are the same type, namely τ_1. We want to give the relation R_τ. A value $a : \tau_1 A.\mathtt{set}$ is related to $b : \tau_1 B.\mathtt{set}$ just when, for every $v : \tau_1$, av is true if and only if a value abstractly equivalent to v appears in the list c. It is the case that R_{τ_1} is abstract equivalence.

Let us turn to the case that τ_1 contains holes. The relation R_{τ_1} will not be simply abstract equivalence. A value $a : \tau_1[A]A.\mathtt{set}$ is related to $b : \tau_1[B]B.\mathtt{set}$ just when, for every related $a_1 : \tau_1[A]$ and $b_1 : \tau_1[B]$, aa_1 is true if and only if b_1 appears in the list b. The definition of $R_{\tau_1[].\mathtt{set}}$ is exactly the same as before: the only thing which has changed is R_{τ_1}.

From this example we can see that we define $R_{(\tau_1,\ldots,\tau_n)[].t}$ by giving a function which takes a relation for each τ_i and returns a relation on $R_{(\tau_1,\ldots,\tau_n)[].t}$. Call this function $R_{[].t}$. Then define:

$$R_{(\tau_1,\ldots,\tau_n)[].t} = R_{[].t} R_{\tau_1} \ldots R_{\tau_p}$$

In order to give a particular family of relations R_τ, it will be necessary to give a function $R_{[].t}$ for every type constructor t appearing in SIG. Of course, when t takes no parameters, $R_{[].t}$ is simply a relation, rather than a function from relations to relations.

8.3.3.2 Type constructors occurring without a hole

Now we move on to the problem of giving $R_{(\tau_1,\ldots,\tau_n)t}$. As an example, we show how to relate $\tau_1[A]\mathtt{list}$ to $\tau_1[B]\mathtt{list}$, for structures A and B, where τ_1 is a type with holes.

First, suppose that τ_1 does not contain any holes. Then $\tau_1[A] = \tau_1 = \tau_1[B]$. We wish to relate $a : \tau_1 t$ to $b : \tau_1 t$ only when a and b are abstractly equivalent.

Now suppose that τ_1 does contain holes. This means that $\tau_1[A]\texttt{list}$ is not the same type as $\tau_1[B]\texttt{list}$. We want to relate $a : \tau_1[A]\texttt{list}$ and $b : \tau_1[B]\texttt{list}$ when they have equivalent "shape," and have related values of types $\tau_1[A]$ and $\tau_1[B]$ at corresponding positions.

As before (when we defined $R_{(\tau_1,\ldots,\tau_n)[].t}$), $R_{(\tau_1,\ldots,\tau_n)t}$ is defined by giving a function which takes a relation for each τ_i and returns a relation on $R_{(\tau_1,\ldots,\tau_n)[].t}$. However, in defining $R_{(\tau_1,\ldots,\tau_n)[].t}$, we define different functions $R_{[].t}$ in order to get different relations R_τ. In defining $R_{(\tau_1,\ldots,\tau_n)t}$, the function depends only on the abstract equivalence for the datatype t. Since it turns out that the function has an identical role in defining Eq_τ to that held by $R_{[].t}$ in defining R_τ, we call this function Eq_t (Eq takes types, not types with holes, hence t rather than $[].t$).

In short, when the type constructor has holes, we take a relation for each parameter type and form whatever relation we like with those relations. When the type constructor has no holes, we again take a relation for each parameter type; we then require that related values have the abstractly equivalent "shapes", and have related values in corresponding places in those shapes.

8.3.4 Polymorphic types

What exactly do we mean when we relate values with polymorphic type? We mean that, for every possible type instantiation, the values are related at that type. This is the idea used, for example, by Wadler [Wad89] in giving logical relations for System F types. (In fact, now that we are considering polymorphism, we are giving relations for type *schemes* rather than types.)

Suppose we wish to give a logical structure relation for a type scheme σ of the form $\forall\alpha.\,\tau$. The relation at α is an arbitrary relation between (possibly different) arbitrary types. Two values of type σ are related only if they are related for any such choice of relation used at α.

That is to say, let A and A' by arbitrary types; let T be an arbitrary relation between A and B. Consider the relation U at τ when the relation at every occurrence of α within τ is T. Two values having type scheme σ are related if and only they are related for every such U.

The definition of logical structure relations for polymorphic type schemes is rather more tentative than the definition for other types. I do not formalize it here, nor do I investigate results for polymorphic types in later sections. It is to be expected that formally defining logical structure relations for polymorphic types will involve detailed analysis and research which is beyond the scope of the present dissertation given its time constraints.

8.4 Definitions

We are now in a position to define logical structure relations, and to formalize the arguments in previous sections. Since the definitions for polymorphic types are considerably more complex than the definitions for monotypes and the definitions for polymorphism are tentative, we only give the definitions for monotypes.

We first define the abstract equivalence relation Eq_τ for all types τ.

8.4.1 Abstract equivalence

$$\begin{array}{lcl}
x\ \mathrm{Eq}_{\{lab_1:\tau_1;\ldots;lab_r:\tau_r\}}\ y & = & (\#lab_1(x)\ \mathrm{Eq}_{\tau_1}\ \#lab_1(y)) \wedge \ldots \\
 & & \wedge(\#lab_r(x)\ \mathrm{Eq}_{\tau_r}\ \#lab_r(y)) \\
x\ \mathrm{Eq}_{\tau_1\rightarrow\tau_2}\ y & = & \forall x_1, y_1\colon \tau_1.\, x_1\ \mathrm{Eq}_{\tau_1}\ y_1 \supset x x_1\ \mathrm{Eq}_{\tau_2}\ y y_1 \\
x\ \mathrm{Eq}_{(\tau_1,\ldots,\tau_n)t}\ y & = & x(\mathrm{Eq}_t \mathrm{Eq}_{\tau_1} \ldots \mathrm{Eq}_{\tau_p})y
\end{array}$$

Every type constructor t must have a function Eq_t defined at the same time as the type constructor is declared.

What properties must Eq_t have? It must, when passed per arguments, return a per result. (In fact, we generalize slightly, and require that when passed arguments which are bijective up to Eq, it returns a result which is bijective up to Eq: as we prove in Chapter 3, the two are equivalent when the relations have types of the form $\tau \twoheadrightarrow \tau \twoheadrightarrow o$). We also require that it does not "look inside" its arguments: it can only use them (or not). This is formalized by the following parametricity constraint.

$$(\mathrm{Eq}_t\ T_1 \ldots T_n) \circ (\mathrm{Eq}_t\ T'_1 \ldots T'_n) = \mathrm{Eq}_t\ (T_1 \circ T'_1) \ldots (T_n \circ T'_n)$$

This says roughly that we can shuffle the relations which apply to elements of parameter types between applications of Eq_t as much as we like.

We define the well-behavedness predicate $WB_\tau(x) = x\ \mathrm{Eq}_\tau\ x$. We require that every visible value in every structure in scope is well-behaved.

8.4.2 Logical structure relations

We define the logical structure relation R between structures S : SIG and S' : SIG, written $R \in \mathit{LSR}(S : \mathrm{SIG}, S')$, as follows.

$$\begin{array}{lcl}
x\ R_{\{lab_1:\tau_1;\ldots;lab_r:\tau_r\}}\ y & = & (\#lab_1(x)\ R_{\tau_1}\ \#lab_1(y)) \wedge \ldots \\
& & \wedge(\#lab_r(x)\ R_{\tau_r}\ \#lab_r(y)) \\
x\ R_{\tau_1 \to \tau_2}\ y & = & \forall x_1 \colon \tau_1[S] | \mathrm{WB}_{\tau_1[S]}. \forall y_1 \colon \tau_1[S'] | \mathrm{WB}_{\tau_1[S']}. \\
& & \quad x_1\ R_{\tau_1}\ y_1 \supset x x_1\ R_{\tau_2}\ y y_1 \\
x\ R_{(\tau_1,\ldots,\tau_n)t}\ y & = & x(\mathrm{Eq}_t R_{\tau_1} \ldots R_{\tau_p})y \\
x\ R_{(\tau_1,\ldots,\tau_n)[].t}\ y & = & x(R_t R_{\tau_1} \ldots R_{\tau_p})y
\end{array}$$

The properties we require of $R_{[].t}$ are similar to those we require of Eq_t: when passed arguments which are bijective up to Eq, it must return a bijection up to Eq. It also has a parametricity constraint, but a slightly different one.

$$\begin{array}{c}
(\mathrm{Eq}_{S.t}\ T_1 \ldots T_n) \circ (R_{[].t}\ T'_1 \ldots T'_n) \circ (\mathrm{Eq}_{S'.t}\ T''_1 \ldots T''_n) \\
= \\
R_{[].t}\ (T_1 \circ T'_1 \circ T''_1) \ldots (T_n \circ T'_n \circ T''_n)
\end{array}$$

This says roughly that we can shuffle the relations which apply to elements of parameter types into and out of $R_{[].t}$ as much as we like.

8.4.3 Abstract equivalence as a logical structure relation

The definitions for R_τ and Eq are obviously very similar. In fact, we can express Eq as a logical structure relation, which we call Eq$'$. Let $\mathrm{Eq}' \in \mathit{LSR}(S : \mathrm{SIG}, S)$ be given by $\mathrm{Eq}'_\tau = \mathrm{Eq}_{\tau[S]}$, where τ is a type with holes over SIG. (Obviously there is an Eq$'$ for every S, but we assume that the structure will be obvious from context).

To see that Eq$'$ is a logical structure relation, we proceed by induction on the structure of the type with holes τ.

Case τ is a labelled record type: the definitions of abstract equivalence and logical structure relations are identical, so the result follows immediately from the induction hypotheses.

Case $\tau = \tau_1 \to \tau_2$: in the definition of Eq_τ we do not require that x_1 and y_1 are well-behaved. However, we do require that $x_1 \mathrm{Eq}_{\tau_1} y_1$, and since Eq_{τ_1} is a per (see lemma 9.2), we have that x_1 and y_1 are well-behaved. The result then follows from the induction hypotheses.

Case $\tau = (\tau_1, \ldots, \tau_n)t$**:** the definitions are identical. The result follows immediately from the induction hypotheses.

Case $\tau = (\tau_1, \ldots, \tau_n)[].t$**:** when we write Eq' for R in the definition of logical structure relations, the definitions become identical. The result follows immediately from the induction hypotheses.

The parametricity constraint on logical structure relations is easily obtained by a double application of the parametricity constraint on Eq.

8.4.4 Reification

Definition 8.1 *A logical structure relation* $R \in LSR(S : \mathrm{SIG}, S')$ *is a reification if and only if, for every value* $v : \tau$ *visible in* SIG, $S.v \; R_\tau \; S'.v$. *(Note that we are treating the type* τ *appearing in* SIG *as a type with holes).*

Chapter 9

An investigation of the properties of reification

In this chapter we investigate (and often prove) results about the reification relation we have defined. Much of the work is given over to understanding the transitivity of reification: we give a non-constructive, set-theoretic proof of transitivity; we believe that this suggests that a constructive proof of transitivity for Simplified ML may exist. The culmination of this investigation is the Modular Abstraction Conjecture (Conjecture 9.9): the conjecture that reification guarantees program equivalence, which is intimately linked to transitivity.

Since our definition of logical structure relations for polymorphic types is tentative, the proofs in this chapter restrict themselves to monomorphic types. This still means that the results are proved for function types and parametric datatypes—both key concepts in reification for Simplified ML—as well as labelled record types.

9.1 Logical structure relations

In this section we prove results which roughly say "if property P holds at datatypes, then it holds at all types". We call such a result P *extension.*

Lemma 9.1 (Extension of bijection-up-to) *Suppose we have a logical structure relation $R \in LSR(S : \mathrm{SIG}, S')$. Further suppose that, for each type constructor t visible in* SIG, *where t takes n type parameters, $R_{[].t} T_1 \ldots T_n$ is a bijection up to* Eq *whenever $T_1 : \tau_1 \twoheadrightarrow \tau_1' \twoheadrightarrow o$, …, $T_n : \tau_n \twoheadrightarrow \tau_n' \twoheadrightarrow o$ are all bijections up to* Eq.

Then R_τ is a bijection up to Eq *for every type with holes over* SIG, *τ.*

Proof: The proof is by induction on the structure of τ.

Case τ is a labelled record type: the result follows simply from the induction hypotheses.

Case $\tau = \tau_1 \dots \tau_n[].t$: the result follows from the property that R returns a bijection-up-to when passed bijections-up-to.

Case $\tau = \tau_1 \dots \tau_n t$: the result follows from the property that Eq_t returns a bijection-up-to when passed bijections-up-to.

Case $\tau = \tau_1 \to \tau_2$: we prove that R_τ is right-consistent, right-exact and total-up-to; the other half of the result follows by symmetric arguments.

Consistency: we assume $y\ \mathrm{Eq}_{\tau[S']}\ y'$ and $x\ R_\tau\ y$. We show that $x\ R_\tau\ y'$.

We expand the definition of $R_{\tau_1 \to \tau_2}$, and then assume $x_1\ R_{\tau_1}\ y_1$ and also $\mathrm{WB}_{\tau_1[S]}(x_1)$ and $\mathrm{WB}_{\tau_1[S']}(y_1)$. It remains to show $xx_1\ R_{\tau_2}\ y'y_1$. From the fact that y_1 is well-behaved and $y\ \mathrm{Eq}_{\tau[S']}\ y'$ we can deduce $yy_1\ \mathrm{Eq}_{\tau_2[S']}\ y'y_1$. From the fact that $x\ R_\tau\ y$, $x_1\ R_{\tau_1}\ y_1$ and the well-behavedness of x_1 and y_1, we can deduce $xx_1\ R_{\tau_2}\ yy_1$. Applying these facts together with right consistency, we obtain that $xx_1\ R_{\tau_2}\ y'y_1$, as required.

Exactness: we assume $x\ R_\tau\ y$ and $x\ R_\tau\ y'$. We must show $y\ \mathrm{Eq}_{\tau[S']}\ y'$.

Assume $y_1\ \mathrm{Eq}_{\tau_1[S']}\ y_1'$ (and thus that y_1 and y_1' are well-behaved). By surjectivity, we can deduce that there is a well-behaved x_1 with $x_1\ R_{\tau_1}\ y_1$. From right-consistency, it follows that $x_1\ R_{\tau_1}\ y_1'$. Applying these facts together with well-behavedness and the facts that $x\ R_\tau\ y$ and $x\ R_\tau\ y'$, we obtain $xx_1\ R_\tau\ yy_1$ and $xx_1\ R_\tau\ y'y_1'$. By right-exactness, $yy_1\ \mathrm{Eq}_{\tau_2[S']}\ y'y_1'$, as required.

Totality-up-to: assume $x : \tau[S]$ is a well-behaved value. We must exhibit a well-behaved y such that $x\ R_\tau\ y$.

Let $f : \tau_1[S'] \twoheadrightarrow \tau_1[S]$ be given by:

$$fy_1 = \varepsilon x_1 {:}\, \tau_1[S].\, \mathrm{WB}_{\tau_1[S]}(x_1) \wedge x_1\ R_{\tau_1}\ y_1$$

Let $g : \tau_2[S] \twoheadrightarrow \tau_2[S']$ be given by:

$$gx_2 = \varepsilon y_2 {:}\, \tau_2[S'].\, \mathrm{WB}_{\tau_2[S']}(y_2) \wedge x_2\ R_{\tau_2}\ y_2$$

Let $y = g \circ x \circ f$. We now show that y is well-behaved and $x\ R_\tau\ y$. Let $y_1 : \tau_1[S']$ be a well-behaved value, and let $a = fy_1$. By surjectivity-up-to, there is a well-behaved x_1 with $x_1\ R_{\tau_1}\ y_1$, so

therefore a is well-behaved and $a\ R_{\tau_1}\ y_1$. Let $b = xa$: since x and a are well-behaved, b is well-behaved. Let $c = gb$. By totality-up-to we can deduce that c is well-behaved and $b\ R_{\tau_2}\ c$. By its construction, we can see that $c = yy_1$, and so y is well-behaved.
Now suppose that $x_1 : \tau_1[S]$ is a well-behaved value, and $x_1\ R_{\tau_1}\ y_1$. By exactness, we can see that $x_1\ \mathrm{Eq}_{\tau_1[S]}\ a$. Since x is well-behaved (and $b = xa$), we have that $xx_1\ \mathrm{Eq}_{\tau_2[S]}\ b$. Since $b\ R_{\tau_2}\ c$ (and $c = yy_1$), by consistency $xx_1\ R_{\tau_2}\ yy_1$. This shows that $x\ R_\tau\ y$, as required.

□

This proof is flawed: it uses the non-constructive operator ε. (This also means that we are using a value with a non-ML type when we expect one with an ML type). This means that we are using values which have no clear connection with ML values in a proof about ML.

However, we should not overstate the case against it: it is entirely set-theoretic, so it is hardly surprising that it does not contain any ML. In these circumstances we can regard the ε operator as the "program" which, in a constructive proof, would be spelled out explicitly in ML.

That this proof is flawed will, in turn, mean that our proof of transitivity is itself flawed. One could attempt to prove this lemma using another method (probably induction on ML expressions). We shall, in section 9.4, suggest a constructive way of proving transitivity which does not use this lemma at all.

Lemma 9.2 (per extension) *Suppose that, for every visible type constructor t, where t takes n type parameters, $\mathrm{Eq}_t T_1 \ldots T_n$ is a per whenever each T_i is a per. Then Eq_τ is a per for every type τ.*

Proof: The simplest proof of this lemma is to combine lemma 3.10 and the previous lemma. There is also a simple constructive proof, which is omitted.

□

9.2 Composition

We now prove some basic results about composition of logical structure relations.

Lemma 9.3 (Identity for composition) *Let $R \in LSR(S : \mathrm{SIG}, S')$ be a reification. Then:*

1. $\mathrm{Eq}' \circ R = R$.

2. $R \circ \mathrm{Eq}' = R$;

Proof: We prove $R_\tau = \mathrm{Eq}'_\tau \circ R_\tau$ and $R_\tau = R_\tau \circ \mathrm{Eq}'_\tau$ for all τ.

The proofs are by induction on the structure of type-with-holes τ.

Case $\tau = \tau_1 \dots \tau_n[].t$: by the definition of *LSR*:

$$(\mathrm{Eq}' \circ R)_\tau = (\mathrm{Eq}' \circ R)_{[].t}(\mathrm{Eq}' \circ R)_{\tau_1} \dots (\mathrm{Eq}' \circ R)_{\tau_n}$$

By the induction hypotheses this is equal to:

$$(\mathrm{Eq}' \circ R)_{[].t} R_{\tau_1} \dots R_{\tau_n}$$

By the definition of composition:

$$\mathrm{Eq}'_{[].t}\mathrm{Eq}'_{\tau_1} \dots \mathrm{Eq}'_{\tau_n} \circ R_{[].t} R_{\tau_1} \dots R\tau_n$$

Folding this up using the definition of *LSR* once more, this is:

$$\mathrm{Eq}_\tau \circ R_\tau$$

Since R_τ is a reification, it is a bijection up to Eq, and it follows that that this simplifies to R_τ, as required.

Case $\tau = \tau_1 \dots \tau_n t$: this follows by identical reasoning to the previous case.

Case τ is a labelled record: since by the induction hypotheses the result holds for each component of the record, it holds for the whole record.

Case $\tau = \tau_1 \twoheadrightarrow \tau_2$: again, the result is a simple consequence of the induction hypotheses.

The proof of the second part of the lemma uses the first part.

Case $\tau = \tau_1 \dots \tau_n[].t$: by the definition of *LSR*, the induction hypotheses and the definition of composition (as in the previous part of the lemma), we arrive at:

$$R_\tau(\mathrm{Eq}'_{\tau_1}) \dots (\mathrm{Eq}'_{\tau_n}) \circ \mathrm{Eq}'_\tau R_\tau \dots R_{\tau_n}$$

This follows by the parametricity condition on $R_{[].t}$ and the first part of the lemma.

Case $\tau = \tau_1 \dots \tau_n t$: again, this follows by identical reasoning to the previous case.

Case τ is a labelled record: since by the induction hypotheses the result holds for each component of the record, it holds for the whole record.

Case $\tau = \tau_1 \twoheadrightarrow \tau_2$: again, the result is a simple consequence of the induction hypotheses.

$\square$

Now we prove that Eq' is not only an identity for composition, but it is also a reification.

Lemma 9.4 (Identity for reification) Eq' *is a reification from S to S for every structure S.*

Proof: All the conditions for reification are immediate consequences of the conditions on Eq.

$\square$

Lemma 9.5 (Associativity of composition) *For $R \in LSR(S : \mathrm{SIG}, S')$, $R' \in LSR(S' : \mathrm{SIG}, S'')$, $R'' \in LSR(S'' : \mathrm{SIG}, S''')$:*

$$(R \circ R') \circ R'' = R \circ (R' \circ R'')$$

Proof: For every type with holes over SIG, τ, we show:

$$((R \circ R') \circ R'')_\tau = (R \circ (R' \circ R''))_\tau$$

The proof is by induction on the structure of τ.

Case $\tau = \tau_1 \dots \tau_n[].t$: the result is derived by equational reasoning.

$$\begin{aligned}
& ((R \circ R')) \circ R'')\tau \\
= \; & ((R \circ R') \circ R'')_{[].t}((R \circ R') \circ R'')_{\tau_1} \dots ((R \circ R') \circ R'')_{\tau_n} \\
= \; & ((R \circ R')_{[].t}\mathrm{Eq}'_{\tau_1} \dots \mathrm{Eq}'_{\tau_n}) \\
& \circ (R''_{[].t}((R \circ R') \circ R'')_{\tau_1} \dots ((R \circ R') \circ R'')_{\tau_n}) \\
= \; & ((R_{[].t}\mathrm{Eq}'_{\tau_1} \dots \mathrm{Eq}'_{\tau_n}) \circ (R'_{[].t}\mathrm{Eq}'_{\tau_1} \dots \mathrm{Eq}'_{\tau_n})) \\
& \circ (R''_{[].t}((R \circ R') \circ R'')_{\tau_1} \dots ((R \circ R') \circ R'')_{\tau_n})
\end{aligned}$$

By the induction hypothesis:

$$\begin{aligned}
&= ((R_{[].t}\mathrm{Eq}'_{\tau_1}\ldots\mathrm{Eq}'_{\tau_n}) \circ (R'_{[].t}\mathrm{Eq}'_{\tau_1}\ldots\mathrm{Eq}'_{\tau_n}))\\
&\quad \circ R''(R\circ(R'\circ R''))_{\tau_1}\ldots(R\circ(R'\circ R''))_{\tau_n}\\
&= (R_{[].t}\mathrm{Eq}'_{\tau_1}\ldots\mathrm{Eq}'_{\tau_n})\\
&\quad \circ((R'_{[].t}\mathrm{Eq}'_{\tau_1}\ldots\mathrm{Eq}'_{\tau_n})\\
&\qquad \circ(R''_{[].t}(R\circ(R'\circ R''))_{\tau_1}\ldots(R\circ(R'\circ R''))_{\tau_n}))\\
&= (R_{[].t}\mathrm{Eq}'_{\tau 1}\ldots\mathrm{Eq}'_{\tau_n})\\
&\quad \circ((R'\circ R'')_{[].t}(R\circ(R'\circ R''))_{\tau_1}\ldots(R\circ(R'\circ R''))_{\tau_n})\\
&= (R\circ(R'\circ R''))_{[].t}(R\circ(R'\circ R''))_{\tau_1}\ldots(R\circ(R'\circ R''))_{\tau_n}\\
&= (R\circ(R'\circ R''))_{\tau}
\end{aligned}$$

Case $\tau = \tau_1 \ldots \tau_n t$: this case is proved by a similar argument to the previous case.

Case $\tau = \tau_1 \twoheadrightarrow \tau_2$: the result follows simply from the induction hypotheses.

Case τ labelled record: again, the result follows simply from the induction hypotheses.

□

9.3 Transitivity and related results

Theorem 9.6 (Horizontal composition) *Let $R_1 \in LSR(S_1 : \mathrm{SIG}_1, S'_1)$ and $R_2 \in LSR(S_2 : \mathrm{SIG}_2, S'_2)$ be reifications. Let SIG_3, $S_3 : \mathrm{SIG3}$, and $S'_3 : \mathrm{SIG}_3$ be given by the following declarations:*

```
signature SIG₃ =
sig
   structure S₁ : SIG₁;
   structure S₂ : SIG₂;
end;

structure S₃ : SIG3 =
struct
   structure S₁ = S₁;
   structure S₂ = S₂;
```

```
end;

structure S'_3 : SIG3 =
struct
   structure S_1 = S'_1;
   structure S_2 = S'_2;
end;
```

Let $\mathrm{Eq}_{S_3.S_1.t} = \mathrm{Eq}_{S_1.t}$ *for every* t *in* SIG_1 *and* $\mathrm{Eq}_{S_3.S_2.t} = \mathrm{Eq}_{S_2.t}$ *for every* t *in* SIG_2. *Let* $\mathrm{Eq}_{S'_3.S_1.t} = \mathrm{Eq}_{S'_1.t}$ *for every* t *in* SIG_1 *and* $\mathrm{Eq}_{S'_3.S_2.t} = \mathrm{Eq}_{S'_2.t}$ *for every* t *in* SIG_2.

Let $R_3 \in \mathit{LSR}(S_3 : \mathrm{SIG}_3, S'_3)$ *be such that* $(R_3)_{[].S_1.t} = (R_1)_{[].t}$ *for every* t *in* SIG_1, *and* $(R_3)_{[].S_2.t} = (R_2)_{[].t}$ *for every* t *in* SIG_2.

Then R_3 *is a reification.*

Proof: We must show:

- S_3 and S'_3 are well-behaved;
- for each value $v : \tau$ visible in SIG_3, $S_3.v\ (R_3)_\tau\ S'_1.v$;
- R_3 is a bijection up to Eq;
- each $(R_3)_{[].t}$ satisfies the parametricity condition.

Each of these is immediate from the fact that both R_1 and R_3 are reifications.

□

The next theorem is key: it says that composition of logical relations accords with ordinary relational composition. It is central to the proof of transitivity of reification, and thus central to the thesis. It is also the property which fails for ordinary logical relations.

Theorem 9.7 (Compositionality) *Suppose we have reifications* $R \in \mathit{LSR}(S : \mathrm{SIG}, S')$ *and* $R' \in \mathit{LSR}(S' : \mathrm{SIG}, S'')$. *Then for all types with holes over* SIG, τ, *and well-behaved* x *and* z:

$$x\ (R \circ R')_\tau\ z = x\ (R_\tau \circ R'_\tau)\ z$$

Proof: The proof is by induction on the structure of τ.

Case $\tau = \tau_1 \twoheadrightarrow \tau_2$: we begin by showing:

$$x\ (R \circ R')_\tau\ z \supset x\ (R_\tau \circ R'_\tau)\ z$$

Assume $x\ (R_\tau \circ R'_\tau)\ z$. This is equivalent to:

$$\begin{array}{l} \exists y. \\ \quad (\forall x_1, y_1 |\, \mathrm{WB}.\, x_1\ R_{\tau_1}\ y_1 \supset xx_1\ R_{\tau_2}\ yy_1) \\ \quad \wedge\ (\forall y_1, z_1 |\, \mathrm{WB}.\, y_1\ R'_{\tau_1}\ z_1 \supset yy_1\ R'_{\tau_2}\ zz_1) \end{array}$$

Let $x_1 : \tau_1[S]$ and $z_1 : \tau_1[S'']$ be well behaved values. Assuming $x_1\ (R \circ R')_{\tau_1}\ z_1$, we must show $xx_1\ (R \circ R')_{\tau_2}\ zz_1$.

By the induction hypothesis, this is equivalent to assuming:

$$\exists y_1.\, x_1\ R_{\tau_1}\ y_1 \wedge y_1\ R'_\tau\ z_1$$

and showing:

$$\exists y_2.\, xx_1\ R_{\tau_2}\ y_2 \wedge y_2\ R'_{\tau_2}\ zz_1$$

We demonstrate that yy_1 is such a y_2.

Since we know $x_1\ R_{\tau_1}\ y_1$, by the assumption we have that $xx_1\ R_{\tau_2}\ yy_1$, as required. Similarly, since we know $y_1\ R'_{\tau_1}\ z_1$, by the assumption we have that $yy_1\ R'_{\tau_2}\ zz_1$, as required.

We have shown that $(R \circ R')_\tau \subseteq (R_\tau \circ R'_\tau)$. By lemma 3.6, $R_\tau \circ R'_\tau$ is a bijection up to Eq. By lemma 9.1 $(R \circ R')_\tau$ is a bijection up to Eq. By lemma 3.5, $(R \circ R')_\tau = (R_\tau \circ R'_\tau)$.

Case τ is a labelled record: this is proved by a simple appeal to the induction hypotheses.

Case $\tau = \tau_1 \ldots \tau_n[].t$: we prove this case by equational reasoning.

$$\begin{array}{ll} & R_\tau \circ R'_\tau \\ = & (R_{[].t} R_{\tau_1} \ldots R_{\tau_n}) \\ & \circ\, (R'_{[].t} R'_{\tau_1} \ldots R'_{\tau_n}) \\ = & ((\mathrm{Eq}'_{[].t} \mathrm{Eq}'_{\tau_1} \ldots \mathrm{Eq}'_{\tau_n}) \circ (R_{[].t} \mathrm{Eq}'_{\tau_1} \ldots \mathrm{Eq}'_{\tau_n}) \circ (\mathrm{Eq}'_{[].t} R_{\tau_1} \ldots R_{\tau_n})) \\ & \circ\, (R'_{[].t} R'_{\tau_1} \ldots R'_{\tau_n}) \\ = & ((R_{[].t} \mathrm{Eq}'_{\tau_1} \ldots \mathrm{Eq}'_{\tau_n}) \circ (\mathrm{Eq}'_{[].t} R_{\tau_1} \ldots R_{\tau_n})) \\ & \circ\, (R'_{[].t} R'_{\tau_1} \ldots R'_{\tau_n}) \\ = & (R_{[].t} \mathrm{Eq}'_{\tau_1} \ldots \mathrm{Eq}'_{\tau_n}) \\ & \circ\, ((\mathrm{Eq}'_{[].t} R_{\tau_1} \ldots R_{\tau_n}) \circ (R'_{[].t} R'_{\tau_1} \ldots R'_{\tau_n})) \end{array}$$

$$
\begin{aligned}
&= (R_{[].t}\mathrm{Eq}'_{\tau_1} \ldots \mathrm{Eq}'_{\tau_n}) \\
&\quad \circ ((\mathrm{Eq}'_{[].t}R_{\tau_1} \ldots R_{\tau_n}) \circ (R'_{[].t}R'_{\tau_1} \ldots R'_{\tau_n}) \circ (\mathrm{Eq}'_{[].t}\mathrm{Eq}'_{\tau_1} \ldots \mathrm{Eq}'_{\tau_n})) \\
&= (R_{[].t}\mathrm{Eq}'_{\tau_1} \ldots \mathrm{Eq}'_{\tau_n}) \\
&\quad \circ (R'_{[].t}(R_{\tau_1} \circ R'_{\tau_1}) \ldots (R_{\tau_n} \circ R'_{\tau_n})) \\
&= (R \circ R')_\tau
\end{aligned}
$$

Case $\tau = t\tau_1 \ldots \tau_n$: the reasoning is similar to the previous case.

□

Why can we prove this result when it does not hold for ordinary logical relations? It is essentially because, in the function type case, we do not try to prove $(R_\tau \circ R'_\tau) \subseteq (R \circ R')_\tau$ directly. Rather we rely on lemma 3.5 and lemma 9.1. (That there are objections to our proof of lemma 9.1 has already been noted).

Corollary 9.8 (Transitivity) *Given two reifications, $R \in LSR(S : \mathrm{SIG}, S')$ and $R' \in LSR(S' : \mathrm{SIG}, S'')$, then $R \circ R' \in LSR(S : \mathrm{SIG}, S'')$ is a reification.*

Proof: We must show:

- S and S'' are well-behaved;
- for each value $v : \tau$ visible in SIG, $S.v\ (R \circ R')_\tau\ S''.v$;
- $R \circ R'$ is a bijection up to Eq;
- each $(R \circ R')_{[].t}$ satisfies the parametricity condition.

Since R is a reification, S_1 must be well-behaved, and since R' is a reification, S_3 is well-behaved. For any value $v : \tau$ visible in SIG, $S.v\ R_\tau\ S'.v$ and $S'.v\ R'_\tau\ S''.v$, so by the previous theorem $S.v\ (R \circ R')_\tau\ S''.v$. We have already noted in the previous proof that $R \circ R'$ is a bijection-up-to.

The proof that the parametricity condition holds is simple, and is omitted.

□

9.4 Modular abstraction

What is the "meaning" of reification? What is the "meaning" of well-behavedness and abstract equivalence? In this section we give a brief overview of another approach to understanding reification.

The ideal equivalence in operational terms is *contextual equivalence* [Gor94]: two terms are equivalent if and only if there is no program which can distinguish them. Can we compare our equivalences with contextual equivalence? Contextual equivalence has types of the form $\tau \twoheadrightarrow \tau \twoheadrightarrow o$, as does abstract equivalence, whereas reification relations have types of the form $\tau[S] \twoheadrightarrow \tau[S'] \twoheadrightarrow o$. If, however, we were to define "contexts with holes," where, as usual in this dissertation, the holes stand for structure names, it would be possible to generalize contextual equivalence and compare it with reification.

In essence, then, one would like to show the following.

Conjecture 9.9 (Modular Abstraction)

1. *Abstractly equivalent values are contextually equivalent.*

2. *Values related by a reification are contextually equivalent.*

Pitts [Pit98] proves a related theorem for System F, and we believe that his proof could be adapted to prove Modular Abstraction.

We can read the conjecture in a number of ways. Firstly, it is a statement about the modules system: it says that modules prevent us from seeing behaviour which differentiates between data which we would consider the same. That is, modules abstract correctly (hence the name of the conjecture).

Secondly, it is a statement about contextual equivalence: it says that there is a simple way to reason about contextual equivalence. This view justifies one of the primary claims for this work: programmers intuitively know the abstract equivalence relation when they write code—they would be unable to program correctly if they did not know which items of data "looked the same". Likewise, when writing a concrete version of abstract code, they intuitively know the reification relation. For example, in Chapter 10, we believe that the abstract equivalance and reification relations should be simple and obvious to most programmers. Writing these relations formally can be done with little effort and gives all the well-known advantages of formal methods: ease of validation, an early trap for bugs, and so on. However, it also gives another major advantage. Proving contextual equivalence is in general is still a (hard) research topic [Gor94]. However, the method outlined in this dissertation is a simple, uniform way to prove the behavioural equivalence of modular functional programs.

Thirdly, it is a statement about reification in ML: it tells us that the reification relation works exactly as we would hope.

Well behavedness relates to "code-ability": by only allowing access to well-behaved values, one would like to be able to ensure that all ML programs are well-behaved. Unfortunately, the converse is not true: not all well-behaved values correspond to ML programs.

Counter-example: There is a well-behaved value in the model which cannot be written as a Simplified ML expression.

The function is that which solves the Halting Problem. Since our model is in classical set theory, it contains all possible (set-theoretic) functions which map Turing Machines to booleans: this includes the function which solves the Halting Problem, H say. Suppose we have a well-behaved representation of a Turing Machine, M (we do no develop the type of Turing Machines and its abstract equivalence here, but it is clear that it could be done). Then HM is either `true` or `false`: both are well-behaved boolean values.

Thus H is a well-behaved function. However, it cannot be coded in Simplified ML, or any effective model of computation, because the Undecidability of the Halting Problem.

A proof that every Simplified ML expression is well-behaved would be by induction on the well-typedness of expressions. Wadler's proof of the Identity Extension Lemma [Wad89] shows how such a proof would proceed.

If the Modular Abstraction Conjecture holds, there is an elegant and simple proof of transitivity.

Outline Proof of Theorem 9.8: The key to this proof is that contextual equivalence across structures is transitive. Suppose $x : \tau[S]$ and $y : \tau[S']$ are contextually equivalent, and that y and $z : \tau[S'']$ are contextually equivalent. Consider an arbitrary context C. The values $C(x)$ and $C(y)$ must be the same, as must $C(y)$ and $C(z)$. Thus the values $C(x)$ and $C(z)$ are the same.

Now, since reification and contextual equivalence agree, reification must also be transitive. $\square$

Chapter 10

Case studies of data reification

In this chapter we give a simple worked example of data reification in Simplified ML: finite sets of integers. We begin with a specification, and then reify that into an implementation with sets represented as lists without repetition. We generate the verification obligations for this reification, and give outline proofs that these obligations hold. We then reify our first implementation into an implementation with sets represented as lists *with* repetition, again generating verification obligations and giving outline proofs.

10.1 A specification

Here is a signature for finite sets of integers in ML:

```
signature SET =
sig
    type Set

    val member: int -> Set -> bool
    val empty: Set
    val insert: int -> Set -> Set
    val remove: int -> Set -> Set
    val inter: Set -> Set -> Set
    val union: Set -> Set -> Set
end
```

The rest of the section develops a specification for a structure of sets with signature `SET`. Data reification is a "model-theoretic" method: meaning that specifications take the form of abstract implementations. The first part of this section gives the axioms which one might use for an algebraic specification of sets. The second part then goes on to give an equivalent model-theoretic specification.

10.1.1 Axioms for sets

Empty sets have no members.

Axiom `Empty`**:**

$\forall x\colon \mathtt{int}.\, \mathsf{isfalse}\,(\mathtt{member}\ \ x\ \ \mathtt{empty})$

Inserting an element into a set makes it an element of that set, but does not change membership of the rest of the set.

Axiom `Insert`**:**

$$
\begin{array}{l}
\forall x, y\colon \mathtt{int}.\, \forall s\colon \mathtt{Set}.\\
\quad \mathsf{istrue}\,(\mathtt{member}\ \ x\ \ (\mathtt{insert}\ \ y\ \ s))\\
\quad\quad \iff\\
\quad (x \approx_{\mathtt{int}} y) \lor \mathsf{istrue}\,(\mathtt{member}\ \ x\ \ s)
\end{array}
$$

This axiom could be split into two.

$$
\begin{array}{l}
\forall x\colon \mathtt{int}.\, \forall s\colon \mathtt{Set}.\\
\quad \mathsf{istrue}\,(\mathtt{member}\ \ x\ \ (\mathtt{insert}\ \ x\ \ s))
\end{array}
$$

$$
\begin{array}{l}
\forall x, y\colon \mathtt{int}.\, \forall s\colon \mathtt{Set}.\, (x \not\approx_{\mathtt{int}} y) \supset\\
\quad \mathsf{istrue}\,(\mathtt{member}\ \ x\ \ (\mathtt{insert}\ \ y\ \ s)) \iff \mathsf{istrue}\,(\mathtt{member}\ \ x\ \ s)
\end{array}
$$

The two forms are equivalent.

Removing an element from a set means it is no longer a member of that set, but does not change membership of the other elements of that set.

Axiom `Remove`**:**

$$
\begin{array}{l}
\forall x, y\colon \mathtt{int}.\, \forall s\colon \mathtt{Set}.\\
\quad \mathsf{istrue}\,(\mathtt{member}\ \ x\ \ (\mathtt{remove}\ \ y\ \ s))\\
\quad\quad \iff\\
\quad (x \not\approx_{\mathtt{int}} y) \land \mathsf{istrue}\,(\mathtt{member}\ \ x\ \ s)
\end{array}
$$

Again, this axiom could be split into two.

$$
\begin{array}{l}
\forall x\colon \mathtt{int}.\, \forall s\colon \mathtt{Set}.\\
\quad \mathsf{isfalse}\,(\mathtt{member}\ \ x\ \ (\mathtt{remove}\ \ x\ \ s))
\end{array}
$$

$$
\begin{array}{l}
\forall x, y\colon \mathtt{int}.\, \forall s\colon \mathtt{Set}.\, (x \not\approx_{\mathtt{int}} y) \supset\\
\quad \mathsf{istrue}\,(\mathtt{member}\ \ x\ \ (\mathtt{remove}\ \ y\ \ s)) \iff \mathsf{istrue}\,(\mathtt{member}\ \ x\ \ s)
\end{array}
$$

The definition of intersection.

Axiom `Inter`**:**

$$\forall x{:}\,\texttt{int}.\,\forall s1, s2{:}\,\texttt{Set}.$$
$$\mathsf{istrue}\,(\texttt{member}\ \ x\ \ (\texttt{inter}\ \ s1\ \ s2))$$
$$\Longleftrightarrow$$
$$\mathsf{istrue}\,(\texttt{member}\ \ x\ \ s1) \wedge \mathsf{istrue}\,(\texttt{member}\ \ x\ \ s2)$$

The definition of union.

Axiom `Union`**:**

$$\forall x{:}\,\texttt{int}.\,\forall s1, s2{:}\,\texttt{Set}.$$
$$\mathsf{istrue}\,(\texttt{member}\ \ x\ \ (\texttt{union}\ \ s1\ \ s2))$$
$$\Longleftrightarrow$$
$$\mathsf{istrue}\,(\texttt{member}\ \ x\ \ s1) \vee \mathsf{istrue}\,(\texttt{member}\ \ x\ \ s2)$$

10.1.2 An abstract implementation

Model-theoretic specifications give an abstract model for the thing being specified (a program, say). It may not be possible to implement the abstract model in a real programming language, or the implementation may be unacceptably inefficient. The implementation process gives steadily more concrete versions of the specified program, until the final program in is reached, written in a real programming language, and suitably efficient.

Algebraic specifications characterize the class of models which implement the specified program. The implementation process makes the program more concrete, and hence reduces the size of the model class, until the program is finished (and the finished program is the only remaining model).

In Simplified ML, a structure takes the rôle of "program". Simplified ML does not have the mathematical machinery necessary to give a specification which is at once clear, and yet still has only a single model.

But it is possible to get around this problem. The abstract implementation of sets, `Set1`, will contain no ML code. In order to ensure that it is an implementation of sets, we will simply assert that it must satisfy our specification axioms for sets.

In an Isabelle implementation, free meta-variables and scheme variables allow us to leave parts of code unimplemented, so our first "implementation" of sets is a structure `Set1` of signature `SET` which is completely unimplemented.

The quantifications in the specification as it stands refer to every element of the type `Set1.Set`, even those which are not well-behaved. But one is only interested in the well-behaved elements of a type. The quantifications should be restricted to the well-behaved part of the type: we should replace $\forall x\colon \texttt{Set}.\, P$ with $\forall x\colon \texttt{Set}|\,\mathrm{WB}.\, P$, (we will assume that all booleans and integers are well-behaved, so the restriction is not necessary for them).

It is also necessary to assume that every visible component of `Set1` is well-behaved.

The final addition which must be made to the specification is some sort of induction rule: all well-behaved elements of the type `Set` must be finitely generated from the values in the signature. The simplest form of the induction axiom is the following.

Axiom `SetInd`:

$$
\begin{aligned}
&\forall P\colon \texttt{Set} \twoheadrightarrow o.\, P(\texttt{empty}) \supset \\
&\quad (\forall s\colon \texttt{Set}|\,\mathrm{WB}.\, \forall x\colon \texttt{int}.\, P(s) \supset \\
&\qquad P(\texttt{insert}\ x\ s)) \supset \\
&\quad \forall s\colon \texttt{Set}|\,\mathrm{WB}.\, P(s)
\end{aligned}
$$

Abstract equality is simply the familiar extensional equality on sets.

$$
\begin{aligned}
&\forall s1, s2\colon \texttt{Set1.Set}|\,\mathrm{WB}. \\
&\quad (\forall x\colon \texttt{int}.\, \texttt{member}\ x\ s1 \approx_{\texttt{bool}} \texttt{member}\ x\ s2) \supset \\
&\qquad s1 \approx_{\texttt{Set}} s2
\end{aligned}
$$

10.1.3 `Set1` is well-behaved

All the axioms for the operations in `Set1` are in terms of their effects on set membership. This means that all the operations map sets with identical members to sets with identical members: that is, the operations are all well behaved.

10.2 Sets as lists without repetition

```
structure Set2: SET =
struct
    datatype 'a Set = empty | SCons of 'a * 'a Set

    fun member x empty = false
      | member x (SCons (y, s)) =
```

```
        if x = y
        then true
        else member x s

    fun insert x s = if member x s then s else SCons (x, s)

    fun remove x empty = empty
      | remove x (SCons (y, s)) =
        if x = y
        then s
        else SCons (y, remove x s)

    fun inter empty           s2 = empty
      | inter (SCons (x, s1)) s2 =
        if member x s2
        then SCons (x, inter s1 s2)
        else inter s1 s2

    fun union empty           s2 = s2
      | union (SCons (x, s1)) s2 =
        if member x s2
        then union s1 s2
        else SCons (x, union s1 s2)
end
```

For this structure the abstract equality for `Set` is simply that two sets are equivalent when they are both free from repetitions and have the same members.

$\lambda s1, s2$: `Set`.
  istrue (`let`
```
    fun repFree empty = true
      | repFree (Scons (x, l)) =
      if member x l then false else repFree l
    in
      repFree s1 andalso repFree s2
    end)
```
  $\wedge$
  $\forall x$: `int`. `member` x $s1 \approx_{\texttt{bool}}$ `member` x $s2$

Since the abstract equivalence is conceptually inside the structure declaration, it can use `SCons` in patterns and expressions.

As an aside, ML is used in this example to code a recursive predicate. This has an advantage over coding it in the meta-logic, namely that Simplified ML is a programming language, and is thus an easier notation in which to write functions than Higher-Order Logic. It also has a disadvantage: since one is limited to working with Simplified ML types (which are all tree-like), one might not be able to express a function as abstractly as one could with a more permissive method of specifying types and their operators.

10.2.1 `Set2` is well-behaved

10.2.1.1 `empty`

The value `empty` has the same members as itself (that is, none), and is repetition-free.

10.2.1.2 `member`

The result boils down to showing the following.

$$s\ \mathrm{Eq}_{\texttt{Set}}\ t \supset (\texttt{member}\ \ x\ \ s) \approx_{\texttt{bool}} (\texttt{member}\ \ x\ \ t)$$

This follows immediately from the definition of $\mathrm{Eq}_{\texttt{Set2.Set}}$.

10.2.1.3 `insert`

The result boils down to the following.

$$s\ \mathrm{Eq}_{\texttt{Set}}\ t \supset (\texttt{insert}\ \ x\ \ s)\ \mathrm{Eq}_{\texttt{Set}}\ (\texttt{insert}\ \ x\ \ t)$$

Let s' be `insert` x s, and let t' be `insert` x t.

Suppose that x is a member of s. Since s is abstractly equivalent to t, this means that x is also a member of t. This means that s' is s, and t' is t. Since we know s and t are abstractly equivalent, the result is immediate.

Now suppose that x is not a member of s: by abstract equivalence, neither can it be a member of t. The value s' is `SCons` $(x,\ s)$ (similarly t'). Since x is not a member of s and s is repetition free, s' is repetition free. By similar reasoning, t' is also repetition free. We can see that, for $y \not\approx_{\texttt{int}} x$, `member` y $s' \approx_{\texttt{bool}}$ `member` y t'. Furthermore, both s' and t' contain x, so they have the same members.

10.2.1.4 `remove`

The result is:

$$s\ \mathrm{Eq}_{\texttt{Set}}\ t \supset (\texttt{remove}\ \ x\ \ s)\ \mathrm{Eq}_{\texttt{Set}}\ (\texttt{remove}\ \ x\ \ t)$$

Let s' be `remove` x s, and let t' be `remove` x t. First, prove that the axiom `Remove` holds for `Set2.Set`. This is a simple induction, and is omitted.

Clearly, `member` x $s' \approx_{\texttt{bool}}$ `false` $\approx_{\texttt{bool}}$ `member` x t'. By the definition of $\mathrm{Eq}_{\texttt{Set2.Set}}$, for all $z \not\approx_{\texttt{int}} x$, `member` z $s' \approx_{\texttt{bool}}$ `member` z t'.

We show that s' is repetition free by induction on the structure of s.

Case `empty`: s' is `empty`, and is thus repetition free.

Case `SCons` (y, s_0): first suppose $x \approx_{\texttt{int}} y$. Then s' is s_0, and is repetition free since `SCons` (y, s_0) is.

Now suppose $x \not\approx_{\texttt{int}} y$. Then s' is `SCons` (y, `remove` x s_0). Expanding the definition of `repFree`, we must show the following.

```
if member y (remove x s0)
then false
else repFree (remove x s0)
```

By the axiom `Remove`, this is equivalent to the following.

```
if member y s0
then false
else repFree (remove x s0)
```

By the fact that s is repetition free, we know that `member` y s_0 is false. Applying the induction hypothesis, we obtain the result.

A similar proof shows that t is repetition free.

10.2.1.5 `inter`

The required result is:

$s\ \mathrm{Eq}_{\texttt{Set}}\ t \supset s'\ \mathrm{Eq}_{\texttt{Set}}\ t'$
$\supset (\texttt{inter}\ \ s\ \ s')\ \mathrm{Eq}_{\texttt{Set}}\ (\texttt{inter}\ \ t\ \ t')$

Let u be `inter` s s' and let v be `inter` t t'.

The proof begins by establishing the truth of the axiom `Inter` for `Set2`. This is a simple induction and is omitted. Using this axiom, it is simple to establish that u has the same membership as v.

The proof that u is repetition free is an induction on s.

Case `empty`: then u is `empty`, and is thus repetition free.

Case `SCons` (x**,** s_0**):** suppose that x is not a member of s'. In that case, u is `inter` s_0 s', which is repetition free by the induction hypothesis.

Now suppose that x is a member of s'. Then u is `SCons(`x`, inter` s_0 s'`)`. By the induction hypothesis `inter` s_0 s' is repetition free. Since s is repetition free, x cannot be a member of s_0, and so, by `Inter`, cannot be a member of `inter` s_0 s'. This means that u must be repetition free.

A similar proof establishes that v is repetition free.

10.2.1.6 `union`

The required result is:

$s \text{ Eq}_{\texttt{Set}}\ t \supset s' \text{ Eq}_{\texttt{Set}}\ t'$
$\supset (\texttt{union}\ s\ s') \text{ Eq}_{\texttt{Set}}\ (\texttt{union}\ t\ t')$

Let u be `union` s s' and let v be `union` t t'.

The proof begins by establishing the truth of the axiom `Union` for `Set2`. This is a simple induction and is omitted. Using this axiom, it is simple to establish that u has the same membership as v.

The proof that u is repetition free is an induction on s.

Case `empty`: then u is s', and is thus repetition free.

Case `SCons` (x**,** s_0**):** suppose that x is a member of s'. Then u is `union` s_0 s', which is repetition free by the induction hypothesis.

Now suppose that x is not a member of s'. Then u is `SCons(`x`, union` s_0 s'`)`. By the induction hypothesis `union` s_0 s' is repetition free. Since s is repetition free, x cannot be a member of s_0, and so, by `Union`, cannot be a member of `union` s_0 s'. This means that u must be repetition free.

A similar proof establishes that v is repetition free.

10.3 Proof that `Set2` reifies `Set1`

The logical relation, $Q \in LSR(\texttt{Set2} : \texttt{SET}, \texttt{Set1})$, via which `Set2` implements `Set1`, is defined inductively by the following rules.

$\texttt{Set2.empty}\ Q_{[].\texttt{Set}}\ \texttt{Set1.empty}$

$\mathrm{WB}_{\texttt{Set2.Set}}(s) \supset \mathrm{WB}_{\texttt{Set1.Set}}(t) \supset (s2\ Q_{[].\texttt{Set}}\ t) \supset \mathsf{isfalse}\ (\texttt{Set2.member}\ \ n\ \ s)$
$\supset \texttt{Set2.SCons}\ \ (n,\ \ s)\ Q_{[].\texttt{Set}}\ \texttt{Set1.insert}\ \ n\ \ t$

Notice that we only specify Q for those elements of `Set1.Set` and `Set2.Set` which satisfy the datatype invariant.

10.3.1 Bijectivity up to Eq

We show this by showing total functionality-up-to in both directions.

Left to right: We must show the following.

$$\begin{aligned}&\forall x{:}\,\texttt{Set2.Set}\,|\,\mathrm{WB}.\ \exists y{:}\,\texttt{Set1.Set}\,|\,\mathrm{WB}.\ \forall y_1{:}\,\texttt{Set1.Set}.\\ &\qquad (x\ Q_{[].\texttt{Set}}\ y_1) \iff (y\ \mathrm{Eq}_{\texttt{Set1.Set}}\ y_1)\end{aligned}$$

The proof is by induction on the structure of x.

Case `Set2.empty`: let y be `Set1.empty`.

- $\supset$ **direction:** it is simple to show that the only possible value for y_1 is `Set1.empty`. This value is well-behaved, and also, therefore, $y\ \mathrm{Eq}_{\texttt{Set1.Set}}\ y_1$.
- $\subset$ **direction:** the only value abstractly equivalent to `Set1.empty` is itself, and it is also related to `Set2.empty`.

Case $\texttt{Set2.SCons}\ \ (n,\ \ x')$: let y be $\texttt{Set1.insert}\ \ n\ \ y'$, where y' is the value that exists by the induction hypothesis.

- $\supset$ **direction:** by the definition of Q and the induction hypothesis, y_1 must be abstractly equivalent to $\texttt{Set1.insert}\ \ n\ \ y'$; but this is y.
- $\subset$ **direction:** by the definition of Eq and the induction hypothesis, y_1 must be related to $\texttt{Set2.SCons}\ \ (n,\ \ x')$; but this is x.

Right to left: the proof is by `SetInd`, and is similar to the previous case.

10.3.2 Value components of `Set2` and `Set1` are related

10.3.2.1 `empty`

The result is immediate from the definition of Q.

10.3.2.2 `member`

The required result simplifies to:

$s \; Q_{[].\texttt{Set}} \; t \supset (\texttt{Set2.member} \;\; y \;\; s) \approx_{\texttt{bool}} (\texttt{Set1.member} \;\; y \;\; t)$

The proof is by induction on s.

Case s is `Set2.empty`. Then the only possible value of t is `Set1.empty`. The result simplifies to `false` $\approx_{\texttt{bool}}$ `false`.

Case s is `Set2.SCons` $(x,\; s_0)$. Suppose $x \approx_{\texttt{int}} y$: then both sides of the equation simplify to `true`. Now suppose $x \not\approx_{\texttt{int}} y$. By the definition of Q and its bijectivity-up-to Eq, there is a t_0 with $s_0 \; Q_{[].\texttt{Set}} \; t_0$, and $t \; \mathrm{Eq}_{\texttt{Set1.Set}}$ `Set1.insert` $x \;\; t_0$. By applying the axioms for `Set1` and the definition of `Set2.member`, the equation simplifies to:

$$\texttt{Set2.member} \;\; y \;\; s_0 \approx_{\texttt{bool}} \texttt{Set1.member} \;\; y \;\; t_0$$

This follows by the induction hypothesis.

10.3.2.3 `insert`

The required result is:

$s \; Q \; t \supset (\texttt{Set2.insert} \;\; y \;\; s) \; Q_{[].\texttt{Set}} \; (\texttt{Set1.insert} \;\; y \;\; t)$

The proof is a case-split on the form of s.

Case s is `Set2.empty`. By the definition of `Set2.insert`, the result becomes:

$$\texttt{Set2.SCons} \;\; (y,\; s) \; Q_{[].\texttt{Set}} \; \texttt{Set1.insert} \;\; y \;\; t$$

This follows from the definition of Q.

Case s is `Set2.SCons` $(x,\; s_0)$. By the definition of Q and bijectivity-up-to, t is `Set1.insert` $x \;\; t_0$, with $s_0 \; Q_{[].\texttt{Set}} \; t_0$.

Case istrue (`Set2.member` y s)**.** By the definition of `Set2.member`, we have that:

$$\mathsf{istrue}\ (\texttt{Set2.member}\ \ y\ \ s_0)$$

By the result for `member`, istrue(`Set1.member` y t_0). By axiom `Insert`, istrue (`Set1.member` y t). By the definition of `Set2.insert` and Axiom `Insert`, the result becomes:

$s\ Q\ t$

This is an assumption.

Case $\mathsf{isfalse}$ (`Set2.member` y s)**.** By the definition of `Set2.insert`:

$\texttt{Set2.SCons}\ (y,\ s)\ Q_{[].\texttt{Set}}\ \texttt{Set1.insert}\ \ y\ \ t$

This follows from $s\ Q_{[].\texttt{Set}}\ t$ (an assumption) by the definition of Q.

10.3.2.4 `remove`

The required result is:

$s\ Q_{[].\texttt{Set}}\ t \supset (\texttt{Set2.remove}\ \ y\ \ s)\ Q_{[].\texttt{Set}}\ (\texttt{Set1.remove}\ \ y\ \ t)$

The proof is by induction on s.

Case s **is** `Set2.empty`**.** The only possible value for t is `Set1.empty`. By the definition of `Set2.remove` and Axiom `Remove`, the result becomes:

$\texttt{Set2.empty}\ Q_{[].\texttt{Set}}\ \texttt{Set1.empty}$

This follows from the definition of Q.

Case s **is** `Set2.SCons` $(x,\ s_0)$**.** By the definition of Q and bijectivity-up-to, t is `Set1.insert` x t_0, with $s_0\ Q_{[].\texttt{Set}}\ t_0$.

Case $y \approx_{\texttt{int}} x$**.** By the definition of `Set2.remove` and the axiom `Remove`, the result becomes:

$\texttt{Set2.remove}\ \ y\ \ s_0\ Q_{[].\texttt{Set}}\ \texttt{Set1.remove}\ \ y\ \ t_0$

This is the induction hypothesis.

Case $y \not\approx_{\texttt{int}} x$**.** By the definition of `Set2.remove` and the axioms for `Set1`, the result becomes:

$\texttt{Set2.SCons}(x,\ \texttt{Set2.remove}\ \ y\ \ s_0)\ Q_{[].\texttt{Set}}\ \texttt{Set1.insert}\ \ x\ \ (\texttt{Set1.remove}\ \ y\ \ t_0)$

By the definition of Q, this follows from:

Set2.remove y s_0 $Q_{[].\texttt{Set}}$ Set1.remove y t_0

This is the induction hypothesis.

10.4 Sets as lists with repetition

```
structure Set3: SET =
struct
    datatype 'a Set = empty | SCons of 'a * 'a Set

    fun insert x s = SCons (x, s)

    fun remove x empty = empty
      | remove x (SCons (y, s)) =
        if x = y
        then remove x s
        else SCons (y, remove x s)

    fun member x empty = false
      | member x (SCons (y, s)) =
        if x = y
        then true
        else member x s

    fun inter empty           s2 = empty
      | inter (SCons (x, s1)) s2 =
        if member x s2
        then SCons (x, inter s1 s2)
        else inter s1 s2

    fun union empty           s2 = s2
      | union (SCons (x, s1)) s2 = SCons (x, union s1 s2)
end
```

The abstract equality for Set is simply extensional equality.

$\lambda s1, s2$: Set.
  $\forall x$: int. member x $s1 \approx_{\texttt{bool}}$ member x $s2$

10.4.1 `Set3` is well-behaved

That `Set3` is well-behaved is a consequence of the fact that it satisfies the set axioms. The proofs necessary to establish this fact are simple inductions, and are omitted.

10.5 Proof that `Set3` reifies `Set2`

In a previous section, we defined the reification $Q \in LSR(\texttt{Set2} : \texttt{SET}, \texttt{Set1})$ with an inductive relation. In this section, we define the reification $R \in LSR(\texttt{Set3} : \texttt{SET}, \texttt{Set2})$ by stating much more directly that elements are related if all possible observations made of the elements are the same. For sets, it is easy to give such a relation: the only possible observations test membership, and R will relate values with identical members.

For other programs it may not be so simple: there may be many possible observations which interact in complex ways. However, where it is possible to give a relation which directly captures observations, this section will show that the proofs become greatly simplified.

We define $R_{[].\texttt{Set}}$ as follows.

$$
\begin{array}{l}
\mathrm{WB}_{\texttt{Set3.Set}}(s) \supset \mathrm{WB}_{\texttt{Set2.Set}}(t) \supset \\
(s\ R_{[].\texttt{Set}}\ t \\
\iff \\
\forall x\colon \texttt{int}.\,(\texttt{Set3.member}\ \ x\ \ s) \approx_{\texttt{bool}} (\texttt{Set2.member}\ \ x\ \ t))
\end{array}
$$

10.5.1 Bijectivity up to Eq

We show this by showing total functionality-up-to in both directions.

Left to right: we must show the following.

$$
\begin{array}{l}
\forall x\colon \texttt{Set3.Set}|\,\mathrm{WB}.\,\exists y\colon \texttt{Set2.Set}|\,\mathrm{WB}.\,\forall y_1\colon \texttt{Set2.Set}. \\
\quad (x\ R_{[].\texttt{Set}}\ y_1) \iff (y\ \mathrm{Eq}_{\texttt{Set2.Set}}\ y_1)
\end{array}
$$

For any given well-behaved x, let y be the `Set2.Set` obtained by removing repetitions from x. Clearly, y and x have the same members, and, since both $R_{[].\texttt{Set}}$ and Eq`Set2.Set` simply compare membership the result holds.

Left to right: the proof is similar; the y that exists is simply x with `Set2.SCons` replaced by `Set3.SCons`.

10.5.2 Value components of `Set3` and `Set2` are related

Simple inductions suffice to establish that the set axioms hold of `Set3`. That the value components are related follows by consideration of the membership of the sets in question.

In order to make clear just how simple these proofs are, we will go through one. The others are left as (tedious) exercises.

10.5.2.1 `inter`

First, by way of an example of the kinds of proof we have been omitting, we establish that the axiom `Inter` does, in fact, hold for `Set3.Set`.

$$\forall x\colon \texttt{int}.\, \forall s1, s2\colon \texttt{Set}.$$
$$\mathsf{istrue}\,(\texttt{member}\ x\ \texttt{(inter}\ s1\ s2\texttt{))}$$
$$\Longleftrightarrow$$
$$\mathsf{istrue}\,(\texttt{member}\ x\ s1) \wedge \mathsf{istrue}\,(\texttt{member}\ x\ s2)$$

The proof is by induction on $s1$.

Case `empty`: both sides of the equivalence are false, by calculation.

Case `SCons(`y`,` s_0`)`: suppose that y is a member of $s2$. Then the left hand side of the equivalence is $\mathsf{istrue}\,(\texttt{member}\ x\ \texttt{(SCons (}y\texttt{, inter}\ s_0\ s2\texttt{)))}$. Further suppose that $x \approx_{\texttt{bool}} y$. Then the left hand side is true. A calculation shows that the right hand side is also true.

Now suppose that y is a member of $s2$, but $x \not\approx_{\texttt{bool}} y$. The left hand side is $\mathsf{istrue}\,(\texttt{member}\ x\ \texttt{(inter}\ s_0\ s2\texttt{))}$. The right hand side asserts that both $\mathsf{istrue}\,(\texttt{member}\ x\ s_0)$ and $\mathsf{istrue}\,(\texttt{member}\ x\ s2)$. Applying the induction hypothesis, the two sides are equivalent.

Now suppose that y is not a member of $s2$ and that $x \approx_{\texttt{bool}} y$. The left hand side is again $\mathsf{istrue}(\texttt{member}\ x\ \texttt{(inter}\ s_0\ s2\texttt{))}$. The right hand side is false. But then, $\mathsf{istrue}\,(\texttt{member}\ x\ s_0) \wedge \mathsf{istrue}\,(\texttt{member}\ x\ s2)$ is also false (and thus equivalent to the right hand side). Applying the induction hypothesis, the two sides are equivalent.

Finally suppose that y is not a member of $s2$ and that $x \not\approx_{\texttt{bool}} y$. The left hand side is again $\mathsf{istrue}\,(\texttt{member}\ x\ \texttt{(inter}\ s_0\ s2\texttt{))}$. The right hand side is again $\mathsf{istrue}\,(\texttt{member}\ x\ s_0) \wedge \mathsf{istrue}\,(\texttt{member}\ x\ s2)$. Applying the induction hypothesis, the two sides are equivalent.

In order to show that Set3.inter $R_{[].\texttt{Set} \to [].\texttt{Set} \to [].\texttt{Set}}$ Set2.inter, we must establish the following.

$\mathrm{WB}_{\texttt{Set3.Set}}(s) \supset \mathrm{WB}_{\texttt{Set3.Set}}(s') \supset \mathrm{WB}_{\texttt{Set2.Set}}(t) \supset \mathrm{WB}_{\texttt{Set2.Set}}(t') \supset$
$s\ R_{[].\texttt{Set}}\ t \supset s'\ R_{[].\texttt{Set}}\ t' \supset$
$(\texttt{Set3.inter}\ \ s\ \ s')\ R_{[].\texttt{Set}}\ (\texttt{Set3.inter}\ \ t\ \ t')$

By using Inter for both Set3 and Set2, this follows immediately.

10.6 Conclusion

In this chapter we have seen reification in action on some simple examples. The work of showing the reifications correct has fallen into two parts in each case.

- First, one must prove some "hygiene" conditions: that all the values in the structure are well-behaved, and that the reification logical structure relation is bijective up to Eq. These conditions are an important validation tool, as well as being necessary for verification: they show that the abstract equivalence for the structures behaves sensibly, as does the reification relation. They can be seen as tests of the abstract equivalence and the reification relation, in that if the conditions fail to hold we should probably change the abstract equivalence or the reification rather than the code.

- Second, we prove the substansive content of the result: that value components of the concrete implementation are related by the reification to those of the abstract implementation. It turns out that, in these examples at least, this amounts to little more than verifying that our original "algebraic" axioms hold for our code.

Our contention in the previous chapter that the programmer intuitively knows the abstract equivalence relation and the reification relation can be carried through to the verification phase: the verifier must check certain properties of the abstract equivalence and the reification, but this amounts to little more than checking that the programmer has written down the relations consistently; once this is done, the proof of correctness is similar to the proof one would perform in an algebraic system.

Chapter 11

Conclusion

> my readers ... will see in the tell-tale compression of the pages before them, that we are all hastening together to perfect felicity.
>
> —Jane Austen, *Northanger Abbey*

11.1 Further work

Part I: Logical Frameworks

Coding binding and substitution explicitly in Isabelle: although we have assessed the use of McKinna-Pollack binding to code logics for programming languages, there exists no such assessment of Stoughton's parallel substitution style. By implementing similar logics to the one given in this chapter in the parallel substitution style, one would be able to directly compare the two methods.

Coding logics in logical frameworks: we claim that the methods we have described in this chapter allow logics to be encoded in a clearer fashion. Our other claim, that these methods increase efficiency, has not been evaluated. Further quantitive work is needed, preferably using implementations of a range of logics, before this claim can be properly assessed.

Part II: Core ML

Simplified ML: although Simplified ML is sufficiently simple to allow us to give a logic for it, it may not be sufficiently powerful to capture real ML programs. It may be necessary to reassess the language in the light of experience.

Reasoning about Core Simplified ML: this logic has not been coded in a theorem prover. Proof experiments with such an implementation may enable us to create an effective tool for program verification.

Part III: Data Reification

Data reification there are many ways of extending the reification system we give. For example: integrating reification of non-terminating programs with the current system; reification of ML functors; reification of programs with local store in the manner of Tennent [Ten94].

An investigation of the properties of reification: the next stage in development of this work would be to prove the Modular Abstraction Conjecture, and thereby give a constructive proof of the transitivity of reification.

Case studies of data reification: although we illustrate the use of our methods in this chapter, it would be necessary to use them for a much larger software project in order to assess their practical usefulness.

11.2 Evaluation

This dissertation has reviewed many aspects of formal methods, from details of machine-assisted proof, to issues in logics for programs, to methods for verification of large programs. It has, therefore, posed many more questions than it has answered. It is our belief, however, that we have addressed the central issues which have arisen.

In particular if we refer to the Hypothesis stated in the Introduction we can see the following.

- We have given new methods for encoding logics for programs in logical frameworks. We have assessed these methods and have provisionally concluded that such encodings can be made clearer and more efficient.

- Using these methods, we have given a logic for Simplified ML programs. Although this logic remains unimplemented, it is comparable with the state of the art in this field.

- We have shown how a variation of data reification can be used to prove Simplified ML programs correct. In particular, we believe that the crucial transitivity property holds.

Appendix A

Isabelle code

> 'What other books did you bind in it?'
>
> 'Tom Paine's *Age of Reason*,' Snead said, consulting his list.
>
> 'What were the results?'
>
> 'Two-hundred-sixty-seven blank pages. Except right in the middle the one word *bleh*.' —Philip K. Dick, *Not by Its Cover*

This appendix contains the Isabelle theory and proof files referred to in Chpater 4.

A.1 The theory files

A.1.1 The main theory

```
(*
    File:  /home/cao/isabelle/mpc/MPHOL.thy
    Theory Name: MPHOL
    Logic Image: HOL
*)

MPHOL  =  closed + distinct +

consts
  NoConns       :: "FList set"
  ValidExts     :: "(FList * FList) set"
  ValidContexts :: "FList set"
  Ctxt          :: "FList => bool"
  WellTypedDecs :: "(FList * FVar Dec) set"
```

```
  IsDec          :: "[FList, FVar Dec] => bool"
  TypesOfF       :: "(FList * FVar * Ty) set"
  LookupTypes    :: "[FList, FVar] => Ty set"
  TypesOfSch     :: "(Scheme * Ty) set"
  TermsOfSch     :: "(Scheme * Term) set"
  TyTermsOfSch   :: "(Scheme * Ty * Term) set"
  InstType       :: "[Scheme, Ty] => bool"
  InstTerm       :: "[Scheme, Term] => bool"
  InstTyTerm     :: "[Scheme, Ty, Term] => bool"
  Judge          :: "(FList * Term) set"
  "@Judge"       :: "[FList, Term] => bool"      ("(4_ |- _)"
                                                 [100, 100] 10)
  TyJudge        :: "(FList * Term * Ty) set"
  "@TyJudge"     :: "[FList, Term, Ty] => bool"  ("(3_ |-/ _:/ _)"
                                               [100, 100, 100] 10)
  G0             :: "[BVar, BVar, BVar, BVar] => FList"


translations
  "tau: LookupTypes (G, f)" == "<G, f, tau>: TypesOfF"
  "Ctxt (G)"                == "G: ValidContexts"
  "IsDec (G, dec)"          == "<G, dec>: WellTypedDecs"
  "InstType (S, tau)"       == "<S, tau>: TypesOfSch"
  "InstTerm (S, M)"         == "<S, M>: TermsOfSch"
  "InstTyTerm (S, tau, M)"  == "<S, tau, M>: TyTermsOfSch"
  "G |- M : tau"            == "<G, M, tau>: TyJudge"
  "G |- P"                  == "<G, P>: Judge"



  (*
   * How declaration schemes are instantiated.
   *)
inductive "TyTermsOfSch"
  intrs
  InstTyTermSchBasic "InstTyTerm (<<tau, M>>, tau, M)"
  InstTyTermSchAbs   "InstTyTerm (S(tau'), tau, M) ==>   \
\       InstTyTerm (SCH A. S(A), tau, M)"
```

```
inductive "TypesOfSch"
  intrs
  InstType           "InstTyTerm (S, tau, M) ==>   \
\                     InstType(S, tau)"


inductive "TermsOfSch"
  intrs
  InstTerm           "InstTyTerm (S, tau, M) ==> InstTerm(S, M)"

  (*
   * Typing judgements.
   *)
inductive "TypesOfF"
  intrs
  LookupVbl  "tau : LookupTypes (G ;; vbl f: tau, f)"
  LookupVal  "InstType(S, tau) ==>   \
\       tau : LookupTypes (G ;; Val (f, S), f)"
  LookupWeak "[| f ~= BoundByDec (dec);   \
\       tau : LookupTypes (G, f) |] ==>   \
\       tau : LookupTypes (G ;; dec, f)"


inductive "TyJudge"
  intrs
  TyLookup  "tau: LookupTypes (G, f) ==>   \
\      G |- $f: tau"
  TyConst   "c HasTy tau ==>   \
\      G |- C$c: tau"
  TyLda     "[| ~ FOccurs (f, M); \
\              (G ;; vbl f: tau) |- M[$f/v]: tau' |] ==>  \
\      G |- lda v: tau. M: tau ->> tau'"
  TyApp     "[| G |- M: tau ->> tau'; G |- N: tau|] ==>   \
\      G |- M ` N: tau'"
```

```
inductive "WellTypedDecs"
  intrs
  IsDecVal "(! tau M. InstTyTerm (S, tau, M) --> \
\               (G |- M: tau)) ==>   \
\     IsDec (G, Val (f, S))"
  IsDecVbl "IsDec (G, vbl f: tau)"

  (*
   * Intuitively, H is a valid extension of the context G if
   * G @@ H is a valid context.
   *)
inductive "ValidExts"
  intrs
  ValidExtsNothing "<G, nothing> : ValidExts"
  ValidExtsDec     "[| <G, H> : ValidExts;  \
\                      IsDec (G @@ H, dec) |] ==>   \
\     <G, H ;; dec> : ValidExts"

  (*
   * G is a valid context if it is a valid extension of
   * the empty context.
   *)
inductive "ValidContexts"
  intrs
  Ctxt    "<nothing, G> : ValidExts ==> Ctxt(G)"


  (*
   * A context has no declarations of connectives.
   *)
inductive "NoConns"
  intrs
  NoConnsNothing "nothing : NoConns"
  NoConnsDec     "[| G : NoConns;   \
\                    ! conn. BoundByDec(dec) ~= D$conn |] ==>   \
\   (G ;; dec) : NoConns"
```

```
rules

  (*
   * Axioms of the formal system.
   *)
  EqRefl     "[| Ctxt (G); G |- M: tau |] ==>   \
\       G |- M === M"

  BetaConv   "[| Ctxt (G);   \
\                G |- (lda b: tau. M) ‘ N : tau’ |] ==>   \
\       G |- ((lda b: tau. M) ‘ N) === M [N/b]"

  OmegaCases "Ctxt (G) ==>   \
\       G |- (forall P: Om. ($$P === T) || ($$P === F))"

  ImpAntiSym "Ctxt (G) ==>   \
\       G |- forall P: Om. forall Q: Om.   \
\                ($$P ---> $$Q) ---> ($$Q ---> $$P)   \
\                      ---> ($$P === $$Q)"

  EtaConv    "Ctxt (G) ==>   \
\       G |- forall H: tau ->> tau’.  \
\                      (lda b: tau. $$H ‘ $$b) === $$H"

  Select     "Ctxt (G) ==>   \
\       G |- forall P: A ->> Om. forall x: A.   \
\             $$P ‘ $$x ---> $$P ‘ (Any ‘ $$P)"


  (*
   * Rules of the formal system.
   *)
(*
 * The McKinna-Pollack formulation of the substitution rule is:
 *
 * Subst      "[| G |- M === M’; G |- P[f=M] |] ==>   \
```

```
 * \       G |- P[f=M']"
 *
 * But due to the nature of FSubst, this can be simplified
 * to the rule below.
 *)
  Subst       "[| G |- M === M'; G |- P(M) |] ==>   \
\       G |- P(M')"

  AbsCong     "[| ~ FOccurs(f, M); ~ FOccurs(f, M');   \
\       (G ;; vbl f: tau ) |- M[$f/b] === N[$f/b'] |] ==> \
\       G |- (lda b: tau. M) === (lda b': tau. N)"

  ImpI        "(G |- P ==> G |- Q) ==>   \
\       G |- P ---> Q"

  MP          "[| G |- P; G |- P ---> Q |] ==>   \
\       G |- Q"

  Lookup      "[| IsDec (G, Val (f, S)); ~ FOccurs(f, M');   \
\                InstTerm(S, M); G |- M === M' |] ==>   \
\       (G ;; Val (f, S)) |- $f === M'"

  Weak        "[| IsDec (G, dec);   \
\          ~FOccurs(BoundByDec(dec), P);   \
\          G |- P |] ==>   \
\       (G ;; dec) |- P"


  (* Definitions of the logical connectives. *)
  G0_def      "G0(P,Q,R,x) ==   \
\     nothing   \
\ ;;   \
\ Val (D$TRUE,   \
\ << Om, ((lda x: A. $$x) === (lda x: A. $$x)) >>)   \
\ ;;   \
\ Val (D$FORALL,   \
\ SCH A. << (A ->> Om) ->> Om,   \
```

```
\ lda P: A ->> Om. ($$P === (lda x: A. T))>>)   \
\ ;;   \
\ Val (D$AND,   \
\ << Om ->> Om ->> Om,   \
\ lda P: Om. lda Q: Om. forall R: Om.   \
\ (($$P ---> $$Q ---> $$R) ---> $$R)>>)  \
\ ;;   \
\ Val (D$OR,   \
\ << Om ->> Om ->> Om,   \
\ lda P: Om. lda Q: Om. forall R: Om.   \
\ (($$P ---> $$R) ---> ($$Q ---> $$R) ---> $$R)>>)  \
\ ;;   \
\ Val (D$FALSE,   \
\ << Om,   \
\ forall x: Om. $$x >>)   \
\ ;;   \
\ Val (D$EXISTS,   \
\ SCH A. << (A ->> Om) ->> Om,   \
\ lda P: A ->> Om. forall R: Om.   \
\ ((forall x: A. $$P ‘ $$x ---> $$R) ---> $$R)>>)   \
\ ;;   \
\ Val (D$NOT,   \
\ << Om ->> Om,   \
\ lda P: Om. ($$P ---> F)>>)"

end

ML
```

A.1.2 B-closedness and other issues

```
(*
    File:  /home/cao/isabelle/mphol/closed.thy
    Theory Name: closed
    Logic Image: HOL
*)

closed  =  bind +
```

```
consts
  Size        :: "Term => nat"
  FFree       :: "[Term] => FVar set"
  FOccurs     :: "[FVar, Term] => bool"
  BOccurs     :: "[BVar, Term] => bool"
  BOccursA    :: "[BVar, Term] => bool"
  BClosed     :: "Term set"
  BClosedA    :: "Term set"
  BClosedB    :: "Term set"
  BClosedC    :: "(Term * BVar set) set"
  BClosedWRT  :: "BVar set => Term set"

translations
  "M: BClosedWRT(s)" == "<M, s> : BClosedC"


  (*
   * The size of a Term.  A slightly odd definition in that the
   * size of atoms is 0.  Has the property that the size of
   * constructed terms is greater than the size of all subterms.
   *)
primrec "Size" bind.Term
  SizeB        "Size ($$b) =   \
\ 0"
  SizeF        "Size ($f') =   \
\ 0"
  SizeC        "Size (C$c) =   \
\ 0"
  SizeApp      "Size (M ` M') =   \
\ Suc(Size (M) + Size (M'))"
  SizeLda      "Size (lda b: tau. M) =   \
\ Suc(Size (M))"

  (*
   * An FVar appears in a Term.
```

```
   *)
primrec "FOccurs" bind.Term
  FOccursB     "FOccurs (f, $$b) =   \
\ False"
  FOccursF     "FOccurs (f, $f') =   \
\ (f = f')"
  FOccursC     "FOccurs (f, C$c) =   \
\ False"
  FOccursApp   "FOccurs (f, M ` M') =   \
\ (FOccurs (f, M) | FOccurs (f, M'))"
  FOccursLda   "FOccurs (f, lda b: tau. M) =   \
\ FOccurs (f, M)"

  (*
   * A BVar appears in a Term.
   *)
primrec "BOccurs" bind.Term
  BOccursB   "BOccurs (b, $$b') =   \
\  (b = b')"
  BOccursF   "BOccurs (b, $f) =   \
\  False"
  BOccursC   "BOccurs (b, C$c) =   \
\  False"
  BOccursApp "BOccurs (b, M ` M') =   \
\  (BOccurs (b, M) | BOccurs (b, M'))"
  BOccursLda "BOccurs (b, lda b': tau. M) =   \
\  if (b = b', False, BOccurs (b, M))"

  (*
   * The free FVars of a Term.
   *)
primrec "FFree" bind.Term
  FFreeB     "FFree ($$b) =   \
\ {}"
  FFreeF     "FFree ($f) =   \
\ {f}"
  FFreeC     "FFree (C$c) =   \
```

```
\ {}"
  FFreeApp   "FFree (M ‘ M’) =   \
\ (FFree (M) Un FFree (M’))"
  FFreeLda   "FFree (lda b: tau. M) =   \
\ FFree (M)"


  (*
   * A Term has no free BVars.
   *)
  inductive "BClosed"
  intrs
  BClosedF     "$f: BClosed"
  BClosedC     "C$c: BClosed"
  BClosedApp   "[| M: BClosed; M’: BClosed |] ==>   \
\ (M ‘ M’): BClosed"
  BClosedLda   "[| M[$f/b]: BClosed; ~FOccurs(f, M) |] ==>   \
\ (lda b: tau. M): BClosed"

  (*
   * A Term has no free BVars.  A different formulation with no
   * side condition on the Lda rule.
   *)
  inductive "BClosedA"
  intrs
  BClosedAF     "$f: BClosedA"
  BClosedAC     "C$c: BClosedA"
  BClosedAApp   "[| M: BClosedA; M’: BClosedA |] ==>   \
\ (M ‘ M’): BClosedA"
  BClosedALda   "M[$f/b]: BClosedA ==>   \
\ (lda b: tau. M): BClosedA"

  (*
   * A Term has no free BVars, except those mentioned.
   *)
  inductive "BClosedC"
  intrs
```

```
  BClosedWRTF   "$f: BClosedWRT(s)"
  BClosedWRTB   "b: s ==>   \
\   $$b: BClosedWRT(s)"
  BClosedWRTC   "C$c: BClosedWRT(s)"
  BClosedWRTApp "[| M: BClosedWRT(s); M': BClosedWRT(s) |] ==> \
\ M ` M': BClosedWRT(s)"
  BClosedWRTLda "[| M[$f/b]: BClosedWRT(s);   \
\                  ~FOccurs(f,M) |] ==>   \
\ lda b: tau. M: BClosedWRT(s)"


rules
  FreshFVar "? f. ~(FOccurs (f, M))"

  BOccursA_def "BOccursA (b, M) ==   \
\       (! f. ~FOccurs (f, M) --> FOccurs (f, M [$f/b]))"

  (*
   * Yet another formulation of BOccurs.  It's an inductive set,
   * but because of the quantifier in the lda rule, it can't be
   * declared as inductive.
   *)
  BClosedB_def "BClosedB ==   \
\        lfp(%X. {z.   \
\        (? f. z = $f) |   \
\        (? c. z = C$c) |   \
\        (? M M'. z = M ` M' & M : X & M' : X) |   \
\        (? M b f tau.   \
\        z = lda b: tau. M &   \
\        (! f. M [$f/b] : X))})"
end
```

A.1.3 Terms

```
(*
    File:  /home/cao/isabelle/mpc/bind.thy
    Theory Name: bind
    Logic Image: HOL
```

```
*)

(*
 * Warning: some priorities, translations etc. are wrong.
 *)
bind  =  Arith +

classes
    Var < term

types
    TyId,
    Id,
    BVar,
    TyVar,
    Scheme 0

arities
    TyId,
    Id,
    Scheme :: term
    BVar :: Var
    TyVar :: Var

datatype
    Ty = Base (TyId)                     ("(T$_)" 210)
    | Fun (Ty, Ty)                       ("(3 _ ->>/ _)"
                                          [111, 110] 110)

datatype
    Const = IMP
    | EQ
    | ANY

datatype
    Conn = TRUE | FALSE | FORALL | EXISTS | NOT | AND | OR
```

```
datatype
    FVar = Def (Conn)                         ("(D$_)" [240] 240)
    | Var (Id)                                ("(V$_)" [240] 240)

datatype
    Term = BV (BVar)                          ("($$_)" [230] 230)
    | FV (FVar)                               ("($_)" [230] 230)
    | Constant (Const)                        ("(C$_)" [220] 220)
    | Lda (BVar, Ty, Term)                    ("(3lda _:/ _./ _)"
                                               [0, 0, 180] 181)
    | App (Term, Term)                        ("(3_ ‘ _)"
                                               [190, 191] 190)

datatype
    'a Dec = Vbl ('a, Ty)                     ("(3vbl _:/ _)" 200)
    | Val ('a, Scheme)

types
    BDec = "BVar Dec"
    FDec = "FVar Dec"

datatype
    BList = BNil
    | BCons (BDec, BList)

datatype
    FList = FNil                              ("nothing")
    | FCons (FList, FDec)                     ("(3_ ;;/ _)"
                                               [120, 121] 120)

datatype
    Renaming = RNil                           ("{||}")
    | RCons (Renaming, FVar, FVar)            ("(3_ ++/ _ |->/ _)"
                                               [120, 121, 121] 120)

consts
  (*
```

```
 * We'd really like Scheme to be a datatype.  There's
 * no (theoretical) reason why it shouldn't be, so we just
 * give the freeness theorems and the induction rule as
 * axioms.
 *)
BasicSch  :: "[Ty, Term] => Scheme"     ("<<_,/ _>>")
AbsSch    :: "(Ty => Scheme) => Scheme" (binder "SCH " 100)

"@val"    :: "[FVar, idts, Ty, Term] => FDec"
                                  ("val _ :/ Sch _./ _ =/ _"
                                        100)
"@@val"   :: "[FVar, Ty, Term] => FDec" ("val _ :/ _ =/ _"
                                        100)

"@Free"   :: "Id => Term"               ("(F$_)" [210] 210)

F         :: "Term"
T         :: "Term"
Forall    :: "Term"
Exists    :: "Term"
Not       :: "Term"
And       :: "Term"
Or        :: "Term"
"@Forall" :: "[BVar, Ty, Term] => Term"
                                       ("(3forall _:/ _./ _)"
                                         [0, 0, 180] 181)
"@Exists" :: "[BVar, Ty, Term] => Term"
                                       ("(3exists _:/ _./ _)"
                                         [0, 0, 180] 181)
"@Not"    :: "Term => Term"             ("(3~~_)" 180)
"@Andd"   :: "[Term, Term] => Term"     ("(3_ &&/ _)" 180)
"@Orr"    :: "[Term, Term] => Term"     ("(3_ ||/ _)" 170)


Imp       :: "Term"
Eq        :: "Term"
Any       :: "Term"
```

```
  "@Imp"    :: "[Term, Term] => Term"     ("(3_ --->/ _)"
                                            [111, 110] 110)
  "@Eq"     :: "[Term, Term] => Term"     ("(3_ ===/ _)"
                                            [100, 101] 100)
  "@Any"    :: "[BVar, Ty, Term] => Term" ("(3any _:/ _./ _)"
                                            [0, 0, 180] 181)

  OMEGA     :: "TyId"
  Om        :: "Ty"

  ConstTy   :: "(Const * Ty) set"
  "@ConstTy":: "[Const, Ty] => bool"      ("_ HasTy/ _")

  BSubst    :: "[Term, Term, BVar] => Term"
                                          ("_ [_'/_]"
                                            [200, 100, 100] 200)
  FSubst    :: "[Term, FVar, Term] => Term"
                                          ("_ [_=_]"
                                            [200, 100, 100] 200)
  RMap      :: "[Renaming, FVar] => FVar"
  Ren       :: "[Renaming, Term] => Term"
  FBoundBy  :: "FList => FVar set"
  BoundByDec:: "'a Dec => 'a"

  FAppend   :: "[FList, FList] => FList"  ("(3_ @@/ _)"
                                            [120, 121] 120)

  (*
   * Substitution for a BVar in a Term.
   *)
primrec "BSubst" Term
  BSubstB    "($$b')[N/b] =   \
\       if (b = b', N, $$b')"
  BSubstF    "($f) [N/b] =   \
\       $f"
  BSubstC    "(C$c) [N/b] =   \
\        C$c"
```

```
  BSubstApp  "(M ‘ M’) [N/b] =   \
\       M[N/b] ‘ M’[N/b]"
  BSubstLda  "(lda b’ : tau. M)[N/b] =   \
\       if (b = b’, lda b’: tau. M, lda b’: tau. M[N/b])"

  (*
   * Substitution for an FVar in a Term.
   *)
primrec "FSubst" Term
  FSubstB    "($$b) [f=N] =   \
\       $$b"
  FSubstF    "($f’) [f=N] =   \
\       if (f = f’, N, $f’)"
  FSubstC    "(C$c) [f=N] =   \
\        C$c"
  FSubstApp  "(M ‘ M’) [f=N] =   \
\       M[f=N] ‘ M’[f=N]"
  FSubstLda  "(lda b: tau. M)[f=N] =   \
\       lda b: tau. M[f=N]"

  (*
   * Applying Renamings to an FVar and to a Term.
   *)
primrec "RMap" Renaming
  RMapNil    "RMap ({||}, f)   \
\       = f"
  RMapCons   "RMap (R ++ f |-> g, f’) =   \
\       if (f = f’, g, RMap (R, f’))"

primrec "Ren" Term
  RenB       "Ren (R, $$b) =   \
\       $$b"
  RenF       "Ren (R, $f) =   \
\       $(RMap (R, f))"
  RenC       "Ren (R, C$c) =   \
\       C$c"
  RenApp     "Ren (R, M ‘ M’) =   \
```

```
\       Ren (R, M) ‘ Ren (R, M’)"
  RenLda     "Ren (R, lda b: tau. M) =   \
\       lda b: tau. Ren (R, M)"

  (*
   * The FVars bound by the declarations in an FList.
   *)
primrec "FBoundBy" FList
  FBoundByFNil  "FBoundBy (nothing) =   \
\  {}"
  FBoundByFCons "FBoundBy (G ;; dec) =   \
\  insert (BoundByDec(dec), FBoundBy (G))"

  (*
   * The variable bound in a declaration.
   *)
primrec "BoundByDec" Dec
  BoundByVbl "BoundByDec (vbl x: tau) =   \
\ x"
  BoundByVal "BoundByDec (Val (x, S)) =   \
\ x"

  (*
   * Appending FLists.
   *)
primrec "FAppend" FList
  FAppendFNil  "G @@ nothing =   \
\ G"
  FAppendFCons "G @@ (G’ ;; d) =   \
\ (G @@ G’) ;; d"


translations
  (*
   * Some of these don’t work as well as intended.
   *)
  "val f : Sch as . tau = M" == "Val (f, SCH as . <<tau, M>>)"
```

```
  "val f : tau = M" == "Val (f, <<tau, M>>)"
  "c HasTy tau"    == "<c, tau>: ConstTy"
  "F$f"            == "$V$f"
  "Eq"             == "C$EQ"
  "Imp"            == "C$IMP"
  "Any"            == "C$ANY"
  "Om"             == "T$OMEGA"
  "x === y"        == "Eq ‘ x ‘ y"
  "P ---> Q"       == "Imp ‘ P ‘ Q"
  "any x: A. P"    == "Any ‘ (lda x: A. P)"
  "F"              == "$D$FALSE"
  "T"              == "$D$TRUE"
  "Forall"         == "$D$FORALL"
  "Exists"         == "$D$EXISTS"
  "Not"            == "$D$NOT"
  "And"            == "$D$AND"
  "Or"             == "$D$OR"
  "forall x: A. P" == "Forall ‘ (lda x: A. P)"
  "exists x: A. P" == "Exists ‘ (lda x: A. P)"
  "~~P"            == "Not ‘ P"
  "P && Q"         == "And ‘ P ‘ Q"
  "P || Q"         == "Or ‘ P ‘ Q"


  (*
   * Types of constants.
   *)
inductive "ConstTy"
  intrs
  TyOfImp "IMP HasTy (Om ->> Om ->> Om)"
  TyOfEq  "EQ  HasTy (tau ->> tau ->> Om)"
  TyOfAny "ANY HasTy ((tau ->> Om) ->> tau)"


rules
  (*
   * These rules make Scheme a datatype.
```

```
   *)
  SchemeDistinct "<<tau, M>> ~= AbsSch(S)"

  BasicInject    "(<<tau, M>> = <<tau', M'>>)   \
\                 = (tau = tau' & M = M')"
  AbsInject      "(AbsSch(S) = AbsSch(S')) = (S = S')"

  SchemeInduct   "[| P(<<tau, M>>);   \
\   !!S. (!A. P(S(A))) ==> P(AbsSch(S)) |] ==>   \
\   P(S')"

end
```

A.1.4 Distinctness

```
(*
    File:  /home/cao/isabelle/mphol/distinct.thy
    Theory Name: distinct
    Logic Image: HOL
*)

distinct  =  List +

consts
  distFrom :: "['a, 'a list] => bool"
  distinct :: "'a list => bool"

primrec "distFrom" List.list
  distFromNil "distFrom (x, []) = True"
  distFromCons "distFrom (x, a # l) =   \
\ ((x ~= a) & (a ~= x) & distFrom (x, l))"

primrec "distinct" List.list
  distinctNil "distinct ([]) = True"
  distinctCons "distinct (a # l) =   \
\ (distFrom(a, l) & distinct (l))"

end
```

A.2 The ML files

A.2.1 The main theory

```
(*
    File:  /home/cao/isabelle/mpc/MPHOL.thy
    Theory Name: MPHOL
    Logic Image: HOL
*)

MPHOL  =  closed + distinct +

consts
  NoConns        :: "FList set"
  ValidExts      :: "(FList * FList) set"
  ValidContexts  :: "FList set"
  Ctxt           :: "FList => bool"
  WellTypedDecs  :: "(FList * FVar Dec) set"
  IsDec          :: "[FList, FVar Dec] => bool"
  TypesOfF       :: "(FList * FVar * Ty) set"
  LookupTypes    :: "[FList, FVar] => Ty set"
  TypesOfSch     :: "(Scheme * Ty) set"
  TermsOfSch     :: "(Scheme * Term) set"
  TyTermsOfSch   :: "(Scheme * Ty * Term) set"
  InstType       :: "[Scheme, Ty] => bool"
  InstTerm       :: "[Scheme, Term] => bool"
  InstTyTerm     :: "[Scheme, Ty, Term] => bool"
  Judge          :: "(FList * Term) set"
  "@Judge"       :: "[FList, Term] => bool"      ("(4_ |- _)"
                                                  [100, 100] 10)
  TyJudge        :: "(FList * Term * Ty) set"
  "@TyJudge"     :: "[FList, Term, Ty] => bool"  ("(3_ |-/ _:/ _)"
                                                [100, 100, 100] 10)
  G0             :: "[BVar, BVar, BVar, BVar] => FList"


translations
  "tau: LookupTypes (G, f)" == "<G, f, tau>: TypesOfF"
```

```
  "Ctxt (G)"                 == "G: ValidContexts"
  "IsDec (G, dec)"           == "<G, dec>: WellTypedDecs"
  "InstType (S, tau)"        == "<S, tau>: TypesOfSch"
  "InstTerm (S, M)"          == "<S, M>: TermsOfSch"
  "InstTyTerm (S, tau, M)"   == "<S, tau, M>: TyTermsOfSch"
  "G |- M : tau"             == "<G, M, tau>: TyJudge"
  "G |- P"                   == "<G, P>: Judge"


  (*
   * How declaration schemes are instantiated.
   *)
inductive "TyTermsOfSch"
  intrs
  InstTyTermSchBasic "InstTyTerm (<<tau, M>>, tau, M)"
  InstTyTermSchAbs   "InstTyTerm (S(tau'), tau, M) ==>   \
\       InstTyTerm (SCH A. S(A), tau, M)"


inductive "TypesOfSch"
  intrs
  InstType           "InstTyTerm (S, tau, M) ==>   \
\                     InstType(S, tau)"


inductive "TermsOfSch"
  intrs
  InstTerm           "InstTyTerm (S, tau, M) ==> InstTerm(S, M)"

  (*
   * Typing judgements.
   *)
inductive "TypesOfF"
  intrs
  LookupVbl  "tau : LookupTypes (G ;; vbl f: tau, f)"
  LookupVal  "InstType(S, tau) ==>   \
\       tau : LookupTypes (G ;; Val (f, S), f)"
```

```
  LookupWeak "[| f ~= BoundByDec (dec);   \
\       tau : LookupTypes (G, f) |] ==>   \
\       tau : LookupTypes (G ;; dec, f)"


inductive "TyJudge"
  intrs
  TyLookup  "tau: LookupTypes (G, f) ==>   \
\      G |- $f: tau"
  TyConst   "c HasTy tau ==>   \
\      G |- C$c: tau"
  TyLda     "[| ~ FOccurs (f, M); \
\              (G ;; vbl f: tau) |- M[$f/v]: tau' |] ==>  \
\      G |- lda v: tau. M: tau ->> tau'"
  TyApp     "[| G |- M: tau ->> tau'; G |- N: tau|] ==>   \
\      G |- M ` N: tau'"


inductive "WellTypedDecs"
  intrs
  IsDecVal "(! tau M. InstTyTerm (S, tau, M) --> \
\              (G |- M: tau)) ==>   \
\     IsDec (G, Val (f, S))"
  IsDecVbl "IsDec (G, vbl f: tau)"

  (*
   * Intuitively, H is a valid extension of the context G if
   * G @@ H is a valid context.
   *)
inductive "ValidExts"
  intrs
  ValidExtsNothing "<G, nothing> : ValidExts"
  ValidExtsDec     "[| <G, H> : ValidExts;  \
\                      IsDec (G @@ H, dec) |] ==>   \
\     <G, H ;; dec> : ValidExts"

  (*
```

```
   * G is a valid context if it is a valid extension of
   * the empty context.
   *)
inductive "ValidContexts"
  intrs
  Ctxt    "<nothing, G> : ValidExts ==> Ctxt(G)"


  (*
   * A context has no declarations of connectives.
   *)
inductive "NoConns"
  intrs
  NoConnsNothing "nothing : NoConns"
  NoConnsDec     "[| G : NoConns;   \
\                    ! conn. BoundByDec(dec) ~= D$conn |] ==>   \
\   (G ;; dec) : NoConns"

rules

  (*
   * Axioms of the formal system.
   *)
  EqRefl     "[| Ctxt (G); G |- M: tau |] ==>   \
\       G |- M === M"

  BetaConv   "[| Ctxt (G);   \
\                G |- (lda b: tau. M) ` N : tau' |] ==>   \
\       G |- ((lda b: tau. M) ` N) === M [N/b]"

  OmegaCases "Ctxt (G) ==>   \
\       G |- (forall P: Om. ($$P === T) || ($$P === F))"

  ImpAntiSym "Ctxt (G) ==>   \
\       G |- forall P: Om. forall Q: Om.   \
\                ($$P ---> $$Q) ---> ($$Q ---> $$P)   \
\                      ---> ($$P === $$Q)"
```

```
  EtaConv    "Ctxt (G) ==>   \
\       G |- forall H: tau ->> tau'.  \
\                     (lda b: tau. $$H ` $$b) === $$H"

  Select     "Ctxt (G) ==>   \
\       G |- forall P: A ->> Om. forall x: A.   \
\            $$P ` $$x ---> $$P ` (Any ` $$P)"


  (*
   * Rules of the formal system.
   *)
(*
 * The McKinna-Pollack formulation of the substitution rule is:
 *
 * Subst      "[| G |- M === M'; G |- P[f=M] |] ==>   \
 * \       G |- P[f=M']"
 *
 * But due to the nature of FSubst, this can be simplified
 * to the rule below.
 *)
  Subst      "[| G |- M === M'; G |- P(M) |] ==>   \
\       G |- P(M')"

  AbsCong    "[| ~ FOccurs(f, M); ~ FOccurs(f, M');   \
\       (G ;; vbl f: tau ) |- M[$f/b] === N[$f/b'] |] ==> \
\       G |- (lda b: tau. M) === (lda b': tau. N)"

  ImpI       "(G |- P ==> G |- Q) ==>   \
\       G |- P ---> Q"

  MP         "[| G |- P; G |- P ---> Q |] ==>   \
\       G |- Q"

  Lookup     "[| IsDec (G, Val (f, S)); ~ FOccurs(f, M');   \
\                 InstTerm(S, M); G |- M === M' |] ==>   \
```

```
\       (G ;; Val (f, S)) |- $f === M'"

  Weak       "[| IsDec (G, dec);   \
\          ~FOccurs(BoundByDec(dec), P);   \
\          G |- P |] ==>   \
\       (G ;; dec) |- P"


  (* Definitions of the logical connectives. *)
  G0_def     "G0(P,Q,R,x) ==   \
\      nothing   \
\ ;;   \
\ Val (D$TRUE,   \
\ << Om, ((lda x: A. $$x) === (lda x: A. $$x)) >>)   \
\ ;;   \
\ Val (D$FORALL,   \
\ SCH A. << (A ->> Om) ->> Om,   \
\ lda P: A ->> Om. ($$P === (lda x: A. T))>>)   \
\ ;;   \
\ Val (D$AND,   \
\ << Om ->> Om ->> Om,   \
\ lda P: Om. lda Q: Om. forall R: Om.   \
\ (($$P ---> $$Q ---> $$R) ---> $$R)>>)  \
\ ;;   \
\ Val (D$OR,   \
\ << Om ->> Om ->> Om,   \
\ lda P: Om. lda Q: Om. forall R: Om.   \
\ (($$P ---> $$R) ---> ($$Q ---> $$R) ---> $$R)>>)  \
\ ;;   \
\ Val (D$FALSE,   \
\ << Om,   \
\ forall x: Om. $$x >>)   \
\ ;;   \
\ Val (D$EXISTS,   \
\ SCH A. << (A ->> Om) ->> Om,   \
\ lda P: A ->> Om. forall R: Om.   \
\ ((forall x: A. $$P ` $$x ---> $$R) ---> $$R)>>)   \
```

```
\ ;;   \
\ Val (D$NOT,   \
\ << Om ->> Om,   \
\ lda P: Om. ($$P ---> F)>>)"

end

ML
```

A.2.2 B-closedness and other issues

```
(*
    File:  /home/cao/isabelle/mphol/closed.thy
    Theory Name: closed
    Logic Image: HOL
*)

closed  =  bind +



consts
  Size        :: "Term => nat"
  FFree       :: "[Term] => FVar set"
  FOccurs     :: "[FVar, Term] => bool"
  BOccurs     :: "[BVar, Term] => bool"
  BOccursA    :: "[BVar, Term] => bool"
  BClosed     :: "Term set"
  BClosedA    :: "Term set"
  BClosedB    :: "Term set"
  BClosedC    :: "(Term * BVar set) set"
  BClosedWRT  :: "BVar set => Term set"

translations
  "M: BClosedWRT(s)" == "<M, s> : BClosedC"



  (*
   * The size of a Term.  A slightly odd definition in that the
```

```
   * size of atoms is 0.  Has the property that the size of
   * constructed terms is greater than the size of all subterms.
   *)
primrec "Size" bind.Term
  SizeB        "Size ($$b) =   \
\ 0"
  SizeF        "Size ($f') =   \
\ 0"
  SizeC        "Size (C$c) =   \
\ 0"
  SizeApp      "Size (M ` M') =   \
\ Suc(Size (M) + Size (M'))"
  SizeLda      "Size (lda b: tau. M) =   \
\ Suc(Size (M))"

  (*
   * An FVar appears in a Term.
   *)
primrec "FOccurs" bind.Term
  FOccursB     "FOccurs (f, $$b) =   \
\ False"
  FOccursF     "FOccurs (f, $f') =   \
\ (f = f')"
  FOccursC     "FOccurs (f, C$c) =   \
\ False"
  FOccursApp   "FOccurs (f, M ` M') =   \
\ (FOccurs (f, M) | FOccurs (f, M'))"
  FOccursLda   "FOccurs (f, lda b: tau. M) =   \
\ FOccurs (f, M)"

  (*
   * A BVar appears in a Term.
   *)
primrec "BOccurs" bind.Term
  BOccursB   "BOccurs (b, $$b') =   \
\  (b = b')"
  BOccursF   "BOccurs (b, $f) =   \
```

```
\  False"
  BOccursC   "BOccurs (b, C$c) =   \
\  False"
  BOccursApp "BOccurs (b, M ‘ M’) =   \
\  (BOccurs (b, M) | BOccurs (b, M’))"
  BOccursLda "BOccurs (b, lda b’: tau. M) =   \
\  if (b = b’, False, BOccurs (b, M))"

  (*
   * The free FVars of a Term.
   *)
primrec "FFree" bind.Term
  FFreeB     "FFree ($$b) =   \
\ {}"
  FFreeF     "FFree ($f) =   \
\ {f}"
  FFreeC     "FFree (C$c) =   \
\ {}"
  FFreeApp   "FFree (M ‘ M’) =   \
\ (FFree (M) Un FFree (M’))"
  FFreeLda   "FFree (lda b: tau. M) =   \
\ FFree (M)"


  (*
   * A Term has no free BVars.
   *)
  inductive "BClosed"
  intrs
  BClosedF     "$f: BClosed"
  BClosedC     "C$c: BClosed"
  BClosedApp   "[| M: BClosed; M’: BClosed |] ==>   \
\ (M ‘ M’): BClosed"
  BClosedLda   "[| M[$f/b]: BClosed; ~FOccurs(f, M) |] ==>   \
\ (lda b: tau. M): BClosed"

  (*
```

```
 * A Term has no free BVars.  A different formulation with no
 * side condition on the Lda rule.
 *)
inductive "BClosedA"
intrs
BClosedAF     "$f: BClosedA"
BClosedAC     "C$c: BClosedA"
BClosedAApp   "[| M: BClosedA; M': BClosedA |] ==>   \
\ (M ` M'): BClosedA"
BClosedALda   "M[$f/b]: BClosedA ==>   \
\ (lda b: tau. M): BClosedA"

(*
 * A Term has no free BVars, except those mentioned.
 *)
inductive "BClosedC"
intrs
BClosedWRTF   "$f: BClosedWRT(s)"
BClosedWRTB   "b: s ==>   \
\   $$b: BClosedWRT(s)"
BClosedWRTC   "C$c: BClosedWRT(s)"
BClosedWRTApp "[| M: BClosedWRT(s); M': BClosedWRT(s) |] ==> \
\ M ` M': BClosedWRT(s)"
BClosedWRTLda "[| M[$f/b]: BClosedWRT(s);   \
\                   ~FOccurs(f,M) |] ==>   \
\ lda b: tau. M: BClosedWRT(s)"



rules
  FreshFVar "? f. ~(FOccurs (f, M))"

  BOccursA_def "BOccursA (b, M) ==   \
\       (! f. ~FOccurs (f, M) --> FOccurs (f, M [$f/b]))"

  (*
   * Yet another formulation of BOccurs.  It's an inductive set,
   * but because of the quantifier in the lda rule, it can't be
```

```
   * declared as inductive.
   *)
  BClosedB_def "BClosedB ==   \
\        lfp(%X. {z.   \
\        (? f. z = $f) |   \
\        (? c. z = C$c) |   \
\        (? M M'. z = M ` M' & M : X & M' : X) |   \
\        (? M b f tau.   \
\        z = lda b: tau. M &   \
\        (! f. M [$f/b] : X))})"
end
```

A.2.3 Terms

```
(*
    File:  /home/cao/isabelle/mpc/bind.thy
    Theory Name: bind
    Logic Image: HOL
*)

(*
 * Warning: some priorities, translations etc. are wrong.
 *)
bind  =  Arith +

classes
    Var < term

types
    TyId,
    Id,
    BVar,
    TyVar,
    Scheme 0

arities
    TyId,
    Id,
```

```
    Scheme :: term
    BVar :: Var
    TyVar :: Var

datatype
    Ty = Base (TyId)                        ("(T$_)" 210)
    | Fun (Ty, Ty)                          ("(3 _ ->>/ _)"
                                             [111, 110] 110)

datatype
    Const = IMP
    | EQ
    | ANY

datatype
    Conn = TRUE | FALSE | FORALL | EXISTS | NOT | AND | OR

datatype
    FVar = Def (Conn)                       ("(D$_)" [240] 240)
    | Var (Id)                              ("(V$_)" [240] 240)

datatype
    Term = BV (BVar)                        ("($$_)" [230] 230)
    | FV (FVar)                             ("($_)" [230] 230)
    | Constant (Const)                      ("(C$_)" [220] 220)
    | Lda (BVar, Ty, Term)                  ("(3lda _:/ _./ _)"
                                             [0, 0, 180] 181)
    | App (Term, Term)                      ("(3_ ‘ _)"
                                             [190, 191] 190)

datatype
    'a Dec = Vbl ('a, Ty)                   ("(3vbl _:/ _)" 200)
    | Val ('a, Scheme)

types
    BDec = "BVar Dec"
    FDec = "FVar Dec"
```

```
datatype
    BList = BNil
    | BCons (BDec, BList)

datatype
    FList = FNil                            ("nothing")
    | FCons (FList, FDec)                   ("(3_ ;;/ _)"
                                             [120, 121] 120)

datatype
    Renaming = RNil                         ("{||}")
    | RCons (Renaming, FVar, FVar)          ("(3_ ++/ _ |->/ _)"
                                             [120, 121, 121] 120)

consts
  (*
   * We'd really like Scheme to be a datatype.  There's
   * no (theoretical) reason why it shouldn't be, so we just
   * give the freeness theorems and the induction rule as
   * axioms.
   *)
  BasicSch  :: "[Ty, Term] => Scheme"     ("<<_,/ _>>")
  AbsSch    :: "(Ty => Scheme) => Scheme" (binder "SCH " 100)

  "@val"    :: "[FVar, idts, Ty, Term] => FDec"
                                 ("val _ :/ Sch _./ _ =/ _"
                                          100)
  "@@val"   :: "[FVar, Ty, Term] => FDec" ("val _ :/ _ =/ _"
                                          100)

  "@Free"   :: "Id => Term"               ("(F$_)" [210] 210)

  F         :: "Term"
  T         :: "Term"
  Forall    :: "Term"
  Exists    :: "Term"
```

```
Not        :: "Term"
And        :: "Term"
Or         :: "Term"
"@Forall" :: "[BVar, Ty, Term] => Term"
                                       ("(3forall _:/ _./ _)"
                                        [0, 0, 180] 181)
"@Exists" :: "[BVar, Ty, Term] => Term"
                                       ("(3exists _:/ _./ _)"
                                        [0, 0, 180] 181)
"@Not"     :: "Term => Term"            ("(3~~_)" 180)
"@Andd"    :: "[Term, Term] => Term"    ("(3_ &&/ _)" 180)
"@Orr"     :: "[Term, Term] => Term"    ("(3_ ||/ _)" 170)


Imp        :: "Term"
Eq         :: "Term"
Any        :: "Term"
"@Imp"     :: "[Term, Term] => Term"    ("(3_ --->/ _)"
                                         [111, 110] 110)
"@Eq"      :: "[Term, Term] => Term"    ("(3_ ===/ _)"
                                         [100, 101] 100)
"@Any"     :: "[BVar, Ty, Term] => Term" ("(3any _:/ _./ _)"
                                         [0, 0, 180] 181)

OMEGA      :: "TyId"
Om         :: "Ty"

ConstTy    :: "(Const * Ty) set"
"@ConstTy":: "[Const, Ty] => bool"      ("_ HasTy/ _")

BSubst     :: "[Term, Term, BVar] => Term"
                                        ("_ [_'/_]"
                                         [200, 100, 100] 200)
FSubst     :: "[Term, FVar, Term] => Term"
                                        ("_ [_=_]"
                                         [200, 100, 100] 200)
RMap       :: "[Renaming, FVar] => FVar"
```

```
  Ren       :: "[Renaming, Term] => Term"
  FBoundBy  :: "FList => FVar set"
  BoundByDec:: "'a Dec => 'a"

  FAppend   :: "[FList, FList] => FList"  ("(3_ @@/ _)"
                                           [120, 121] 120)

  (*
   * Substitution for a BVar in a Term.
   *)
primrec "BSubst" Term
  BSubstB    "($$b')[N/b] =   \
\       if (b = b', N, $$b')"
  BSubstF    "($f) [N/b] =   \
\       $f"
  BSubstC    "(C$c) [N/b] =   \
\        C$c"
  BSubstApp  "(M ` M') [N/b] =   \
\       M[N/b] ` M'[N/b]"
  BSubstLda  "(lda b' : tau. M)[N/b] =   \
\       if (b = b', lda b': tau. M, lda b': tau. M[N/b])"

  (*
   * Substitution for an FVar in a Term.
   *)
primrec "FSubst" Term
  FSubstB    "($$b) [f=N] =   \
\       $$b"
  FSubstF    "($f') [f=N] =   \
\       if (f = f', N, $f')"
  FSubstC    "(C$c) [f=N] =   \
\        C$c"
  FSubstApp  "(M ` M') [f=N] =   \
\       M[f=N] ` M'[f=N]"
  FSubstLda  "(lda b: tau. M)[f=N] =   \
\       lda b: tau. M[f=N]"
```

```
  (*
   * Applying Renamings to an FVar and to a Term.
   *)
primrec "RMap" Renaming
  RMapNil    "RMap ({||}, f)   \
\       = f"
  RMapCons   "RMap (R ++ f |-> g, f') =   \
\       if (f = f', g, RMap (R, f'))"

primrec "Ren" Term
  RenB       "Ren (R, $$b) =   \
\       $$b"
  RenF       "Ren (R, $f) =   \
\       $(RMap (R, f))"
  RenC       "Ren (R, C$c) =   \
\       C$c"
  RenApp     "Ren (R, M ` M') =   \
\       Ren (R, M) ` Ren (R, M')"
  RenLda     "Ren (R, lda b: tau. M) =   \
\       lda b: tau. Ren (R, M)"

  (*
   * The FVars bound by the declarations in an FList.
   *)
primrec "FBoundBy" FList
  FBoundByFNil  "FBoundBy (nothing) =   \
\  {}"
  FBoundByFCons "FBoundBy (G ;; dec) =   \
\  insert (BoundByDec(dec), FBoundBy (G))"

  (*
   * The variable bound in a declaration.
   *)
primrec "BoundByDec" Dec
  BoundByVbl "BoundByDec (vbl x: tau) =   \
\ x"
  BoundByVal "BoundByDec (Val (x, S)) =   \
```

```
\ x"

  (*
   * Appending FLists.
   *)
primrec "FAppend" FList
  FAppendFNil  "G @@ nothing =   \
\ G"
  FAppendFCons "G @@ (G' ;; d) =   \
\ (G @@ G') ;; d"


translations
  (*
   * Some of these don't work as well as intended.
   *)
  "val f : Sch as . tau = M" == "Val (f, SCH as . <<tau, M>>)"
  "val f : tau = M" == "Val (f, <<tau, M>>)"
  "c HasTy tau"    == "<c, tau>: ConstTy"
  "F$f"            == "$V$f"
  "Eq"             == "C$EQ"
  "Imp"            == "C$IMP"
  "Any"            == "C$ANY"
  "Om"             == "T$OMEGA"
  "x === y"        == "Eq ` x ` y"
  "P ---> Q"       == "Imp ` P ` Q"
  "any x: A. P"    == "Any ` (lda x: A. P)"
  "F"              == "$D$FALSE"
  "T"              == "$D$TRUE"
  "Forall"         == "$D$FORALL"
  "Exists"         == "$D$EXISTS"
  "Not"            == "$D$NOT"
  "And"            == "$D$AND"
  "Or"             == "$D$OR"
  "forall x: A. P" == "Forall ` (lda x: A. P)"
  "exists x: A. P" == "Exists ` (lda x: A. P)"
  "~~P"            == "Not ` P"
```

```
  "P && Q"         == "And ‘ P ‘ Q"
  "P || Q"         == "Or ‘ P ‘ Q"



  (*
   * Types of constants.
   *)
inductive "ConstTy"
  intrs
  TyOfImp "IMP HasTy (Om ->> Om ->> Om)"
  TyOfEq  "EQ  HasTy (tau ->> tau ->> Om)"
  TyOfAny "ANY HasTy ((tau ->> Om) ->> tau)"



rules
  (*
   * These rules make Scheme a datatype.
   *)
  SchemeDistinct "<<tau, M>> ~= AbsSch(S)"

  BasicInject    "(<<tau, M>> = <<tau’, M’>>)   \
\                 = (tau = tau’ & M = M’)"
  AbsInject      "(AbsSch(S) = AbsSch(S’)) = (S = S’)"

  SchemeInduct   "[| P(<<tau, M>>);   \
\   !!S. (!A. P(S(A))) ==> P(AbsSch(S)) |] ==>   \
\   P(S’)"

end
```

A.2.4 Distinctness

```
(*
    File:  /home/cao/isabelle/mphol/distinct.thy
    Theory Name: distinct
    Logic Image: HOL
*)
```

```
distinct  =  List +

consts
  distFrom :: "['a, 'a list] => bool"
  distinct :: "'a list => bool"

primrec "distFrom" List.list
  distFromNil "distFrom (x, []) = True"
  distFromCons "distFrom (x, a # l) =   \
\ ((x ~= a) & (a ~= x) & distFrom (x, l))"

primrec "distinct" List.list
  distinctNil "distinct ([]) = True"
  distinctCons "distinct (a # l) =   \
\ (distFrom(a, l) & distinct (l))"

end
```

Bibliography

[Abr96] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[AGNvS94] Thorsten Altenkirch, Veronica Gaspes, Bengt Nordström, and Björn von Sydow. *A User's Guide to ALF.* Chalmers University of Technology, Sweden, May 1994.

[AHM87] Arnon Avron, Furio A. Honsell, and Ian A. Mason. Using typed lambda calculus to implement formal systems on a machine. In Peter Dybjer, Bengt Nordström, Kent Petersson, and Jan M. Smith, editors, *Proceeding of the Workshop on Programming Logic*, pages 336–373, October 1987.

[Alt93] Thorsten Altenkirch. *Constructions, Inductive Types and Strong Normalization.* PhD thesis, Department of Computer Science, The University of Edinburgh, 1993. Published as CST-106-93; also published as LFCS Technical Report ECS-LFCS-93-279.

[ASM79] J.-R. Abrial, S.A. Schuman, and B. Meyer. *Semantics of Concurrent Computation: Proceedings, Evian, France*, volume 70, chapter Specification Language, pages 34–50. Springer-Verlag, Berlin, 1979.

[Avr87] Arnon Avron. Simple consequence relations. Technical report, Laboratory for Foundations of Computer Science, Department of Computer Science, The University of Edinburgh, The King's Buildings, Edinburgh EH9 3JZ, 1987.

[Ber91] Dave Berry. The Edinburgh SML library. Technical Report ECS-LFCS-91-148, Laboratory for Foundations of Computer Science, Department of Computer Science, The University of Edinburgh, The King's Buildings, Edinburgh EH9 3JZ, April 1991.

[BG81] R.M. Burstall and J.A. Goguen. *An Informal Introduction to Specifications Using CLEAR*, pages 185–214. International Lecture Series in Computer Science. Academic Press, London, 1981.

[BGG+92] R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. Van Tassel. Experience with embedding hardware description languages in HOL. In *Theorem Provers in Circuit Design: Theory, Practice and Experience*, 1992.

[Bis67] Errett Bishop. *Foundations of Constructive Analysis.* McGraw-Hill, New York, 1967. Currently out of print.

[BM79] R. S. Boyer and J. S. Moore. *A Computational Logic.* Academic Press, New York, 1979.

[BSvH+93] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993.

[Bun91] Alan Bundy. A science of reasoning. In Jean-Louis Lassez and Gordon Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson.* The MIT Press, 1991.

[Bur87] R.M. Burstall. Inductively defined functions in functional programming languages. *Journal of Computer and System Sciences*, 34(2/3):409–421, 1987.

[BvHHS90] Alan Bundy, Frank van Harmelen, Christian Horn, and Alan Smaill. The Oyster-Clam system. In M. E. Stickel, editor, *Proceedings of the 10th International Conference on Automated Deduction*, volume 449 of *LNAI*, pages 647–648, Kaiserslautern, FRG, July 1990. Springer Verlag.

[C+86] R.L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, Englewood Cliffs, New Jersey 07632, 1986.

[CF58] Haskell B. Curry and Robert Feys. *Combinatory Logic*, volume I. North-Holland, Amsterdam, 1958.

[CH86] Thierry Coquand and Gérard Huet. The Calculus of Constructions. Rapport de Recherche 530, INRIA, Rocquencourt, France, May 1986.

[Chu40] Alonzo Church. A formulation of the Simple Theory of Types. *The Journal of Symbolic Logic*, 5:56–68, 1940.

[Chu65a] Church. A note on the Entscheidungsproblem. In Martin Davis, editor, *The Undecidable*. Raven Press, 1965.

[Chu65b] Church. An unsolvable problem of elementary number theory. In Martin Davis, editor, *The Undecidable*. Raven Press, 1965.

[CJ91] J.H. Cheng and C.B. Jones. On the useability of logics which handle partial functions. In Carroll Morgand and J.C.P. Woodcock, editors, *3rd Refinement Workshop*, Workshops in Computing, pages 51–69. Springer-Verlag, 1991.

[CO92] A. Cant and M.A. Ozols. A verification environment for ML programs. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, June 1992.

[Coh89] Avra Cohn. The notion of proof in hardware verification. *Journal of Automated Reasoning*, 5(2):127–139, June 1989.

[Coq92] Thierry Coquand. Pattern matching with dependent types. In Bengt Nordström, Kent Petersson, and Gordon Plotkin, editors, *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, June 1992.

[dB72] N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem. *Indagationes Mathematicae*, 34(5):381–392, 1972.

[dB80] N. G. de Bruijn. A survey of the project AUTOMATH. In J. R. Hindley and J. P. Seldin, editors, *Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 580–606. Academic Press, London, 1980.

[DFH+93] Dowek, Felty, Herbelin, Huet, Murthy, Parent, Pauling-Mohring, and Werner. The Coq proof assistant user's guide, version 5.8. Technical report, INRIA Rocquencourt, 1993.

[DFH95] Jöelle Despeyroux, Amy Felty, and André Hirschowitz. Higher-order abstract syntax in Coq. In *Typed Lambda Calculi and Applications,* *TLCA '95*, number 902 in Lecture Notes in Computer Science, pages 124–138, Edinburgh, 1995. Springer-Verlag.

[Dum77] Michael Dummet. *Elements of Intuitionism.* Oxford University Press, Oxford, 1977.

[Fef94] Saul Feferman. Finitary inductive systems. In *What is a Logical System?* Oxford University Press, 1994.

[FF90] Simon Finn and Michael P. Fourman. *Logic Manual For The LAMBDA System Version 3.2.* Abstract Hardware Limited, 1990.

[FF91] Simon Finn and Michael P. Fourman. *Logic Manual For The LAMBDA System Version 4.0.* Abstract Hardware Limited, March 1991.

[FFL97] Simon Finn, Michael P. Fourman, and John Longley. Partial functions in a total setting. *Journal of Automated Reasoning*, 18(1):85–104, February 1997.

[Flo67] R.W. Floyd. Assigning meaning to programs. In *Proc. Symp. in Applied Mathematics, Vol. 19: Mathematical Aspects of Computer Science*, pages 19–32. American Mathematical Society, 1967.

[GM93] M.J.C. Gordon and T.F. Melham. *Introduction to HOL.* Cambridge University Press, Cambridge, 1993.

[Goo93] Kees G. W. Goossens. *Embedding Hardware Description Languages in Proof Systems.* PhD thesis, The University of Edinburgh, 1993.

[Gor93] Andrew D. Gordon. A mechanisation of name-carrying syntax up to alpha-conversion. In *HUG'93 Higher Order Logic Theorem Proving and its Applications, Vancouver*, Lecture Notes in Computer Science. Springer-Verlag, 1993.

[Gor94] Andrew D. Gordon. *Functional Programming and Input/Output.* Cambridge University Press, Cambridge, 1994.

[GW88] J.A. Goguen and T. Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI Computer Science Laboratory, 1988.

[HHP87] R.W. Harper, F.A. Honsell, and G.D. Plotkin. A framework for defining logics. In *Proc. 2nd IEEE Symp. on Logic in Computer Science*, Cornell, 1987.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *CACM*, 12:576–580, October 1969.

[Hoa72] C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[Hof92] Martin Hofmann. Formal development of functional programs in type theory—a case study. Technical Report ECS-LFCS-92-228, Laboratory for Foundations of Computer Science, Department of Computer Science, The University of Edinburgh, The King's Buildings, Edinburgh EH9 3JZ, August 1992. Adapted from Master's thesis.

[HS86] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and The Lambda-Calculus*. London Mathematical Society Student Texts. Cambridge University Press, Cambridge, 1986.

[HT95] Steve Hill and Simon Thompson. Miranda in isabelle. In Lawrence C. Paulson, editor, *Proceedings of the Isabelle Users Workshop*, 1995.

[JM92] C.B. Jones and A.M. McCauley. Formal methods - selected historical references. Technical Report UMCS-92-12-2, University of Manchester Department of Computer Science, 1992.

[Jon73] C.B. Jones. Formal development of programs. Technical Report 12.117, IBM Laboratory Hursley, 1973.

[Jon86] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1986.

[Jon92] C.B. Jones. The search for tractable ways of reasoning about programs. Technical Report UMCS-92-4-4, University of Manchester Department of Computer Science, 1992.

[Kah93] Stefan Kahrs. Mistakes and ambiguities in the Definition of Standard ML. Technical Report ECS-LFCS-93-257, Laboratory for Foundations of Computer Science, University of Edinburgh, April 1993.

[Kle98] Thomas Kleymann. *Hoare Logic and VDM: Machine-Checked Soundness and Completeness Proofs*. PhD thesis, The University of Edinburgh, 1998.

[KST94] Stefan Kahrs, Donald Sannella, and Andrzej Tarlecki. The definition of Extended ML. Technical Report ECS-LFCS-94-300, Laboratory for Foundations of Computer Science, University of Edinburgh, August 1994.

[Lak76] Imre Lakatos. *Proofs and Refutations*. Cambridge University Press, Cambridge, 1976.

[Luo89] Z. Luo. ECC, the Extended Calculus of Constructions. In *Logic in Computer Science*, pages 386–395. IEEE, 1989.

[McM92] K.L. McMillan. The SMV system, symbolic model checking - an approach. Technical Report CMU-CS-92-131, Carnegie Mellon University, 1992.

[Mil72] Robin Milner. Logic for computable functions: Description of a machine implementation. Technical Report STAN-CS-72-288, Stanford University, 1972.

[MJ84] F.L. Morris and C.B. Jones. An early program proof by Alan Turing. *Annals of the History of Computing*, 6:139–143, April 1984.

[ML75] P. Martin-Löf. An intuitionistic theory of types: Predictive part. In H. E. Rose and J. C. Sheperdson, editors, *Logic Colloquium '73*, volume 80 of *Studies in Logic*, pages 73–118. North-Holland Publishing Co., New York, N.Y., 1975.

[ML80] P. Martin-Löf. *Intuitionistic type theory*. Bibliopolis, Napoli, 1980.

[MLP79] Richard De Millo, Richard Lipton, and Alan Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):271–280, May 1979.

[MP93] James McKinna and Robert Pollack. Pure type systems formalized. In M. Bezem and J.F. Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications, TLCA'93*, number 664 in Lecture Notes in Computer Science, pages 289–305, Utrecht, March 1993. Springer-Verlag.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts 02142, 1990.

[OSR95] S. Owre, N. Shankar, and J. M. Rushby. *User Guide for the PVS Specification and Verification System*. SRI Computer Science Laboratory, 1995.

[Owe95] Christopher Owens. Coding binding and substitution explicitly in Isabelle. In Lawrence C. Paulson, editor, *Proceedings of the Isabelle Users Workshop*, 1995.

[Pau90] Lawrence C. Paulson. A formulation of the Simple Theory of Types (for Isabelle). In P. Martin-Löf and G. Mints, editors, *COLOG-88: International Conference on Computer Logic*, number 417 in Lecture Notes in Computer Science, pages 246–274. Springer-Verlag, 1990.

[Pau94a] Lawrence C. Paulson. *Isabelle — A Generic Theorem Prover*. Number 828 in Lecture Notes in Computer Science. Springer-Verlag, 1994.

[Pau94b] Lawrence C. Paulson. *The Isabelle Reference Manual*. University of Cambridge Computer Laboratory, 1994.

[PE88] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, 1988.

[Per84] Charles Perrow. *Normal Accidents*. Basic Books, New York, 1984.

[Pit98] A. M. Pitts. Parametric polymorphism and operational equivalence. Technical Report 453, Cambridge University Computer Laboratory, 1998. A preliminary version appeared in *Proceedings, Second Workshop on Higher Order Operational Techniques in Semantics (HOOTS II), Stanford CA, December 1997*, Electronic Notes in Theoretical Computer Science 10, 1998.

[PL92] Robert Pollack and Zhaohui Luo. LEGO proof development system: User's manual. Technical Report ECS-LFCS-92-211, Laboratory for Foundations of Computer Science, Department of Computer Science, The University of Edinburgh, The King's Buildings, Edinburgh EH9 3JZ, May 1992.

[Pol96] Robert Pollack. What we learn from formal checking part I: How to believe a machine-checked proof. Technical Report BRICS Notes Series NS-96-8, BRICS, 1996.

[Rey83] J.C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing 83*, 1983.

[RS92] Mark Ryan and Martin Sadler. Valuation systems and consequence relations. In S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 1. Oxford University Press, 1992.

[San91] Donald Sannella. Formal program development in Extended ML for the working programmer. In Carroll Morgan and J.C.P. Woodcock, editors, *3rd Refinement Workshop*, Workshops in Computing, pages 99–130. Springer-Verlag, January 1991. Also published as LFCS Technical Report ECS-LFCS-90-106.

[Sco79] Dana Scott. Identity and existence in intuitionistic logic. In M.P. Fourman, C.J. Mulvey, and D.S. Scott, editors, *Applications of Sheaves*, number 753 in Lecture Notes in Mathematics, pages 660–696. Springer-Verlag, 1979.

[Sha94] N. Shankar. *Metamathematics, Machines, and Gödel's Proof*, volume 38 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1994.

[Sta85] R. Statman. Logical relations and the typed λ-calculus. *Information and Computation*, 1985.

[Sto88] Allen Stoughton. Substitution revisited. *Theoretical Computer Science*, 59:317–325, 1988.

[SW83] D.T. Sannella and M. Wirsing. A kernel language for algebraic specification and implementation. In *Proc. 1983 Intl. Conf. on Foundations of Computation Theory*, volume LNCS 158, pages 413–427, Borgholm, Sweden, 1983. Springer-Verlag.

[Ten94] R. D. Tennent. Correctness of data representations in Algol-like languages. In W. Roscoe, editor, *A Classical Mind, Essays in Honour of C. A. R. Hoare*, pages 405–417. Prentice-Hall International, 1994. Also published as LFCS Technical Report ECS-LFCS-93-267.

[Tur36] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936. Correction published: ibid, 43:544–546, 1937.

[Tur49] Alan M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, Cambridge, June 1949. University Mathematical Laboratory.

[Wad89] P. Wadler. Theorems for free! *Functional Programming Languages and Computer Architectures*, 1989.

[Wos96] Larry Wos. *The Automation of Reasoning: An Experimenter's Notebook with OTTER Tutorial.* Academic Press, 1996.

[WR10] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica.* Cambridge University Press, Cambridge, 1910. Second edition 1925.

Get stewed:
Books are a load of crap.

—Philip Larkin, *A Study of Reading Habits*

Communism is Soviet power plus the electrification of the whole country.

—Lenin