Machine Assisted Reasoning About Standard ML Using HOL

Donald Syme

A subthesis submitted to the Australian National University in partial fulfillment of the requirements for the degree of Bachelor of Science (Honours)

November 1992

Abstract

This thesis is concerned with aspects of formal reasoning over constructs of the programming language Standard ML. The focus of the work is on the mechanization of the reasoning process using the HOL logic system. The work is a continuation of work begun by Matthew Hutchins at ANU in 1990.

While the main aim of the thesis is to describe the construction of a HOL theory of Standard ML, many other issues are encountered. The problem of formally defining mutually recursive type structures in HOL is addressed, as is the need to develop HOL theories of fundamental constructs such as finite maps. Previous HOL representations of the syntax and semantics of Standard ML are found to be incomplete at several crucial points. The correctness of the representation developed in this projectat is demonstrated by the derivation several important language properties, most of which did not hold for the incorrect representations. Some example applications of the theory the the areas of program verification and program transformations are presented. A recurring theme in the thesis is the need for logical precision in all aspects of a HOL representation.

Acknowledgments

During this and previous years many people have shown me attention and concern beyond any expectation I could make of them, and beyond anything I could hope to return. I would particularly like to thank my family and Ramona O'Brien in this regard. I would also like to thank my supervisor Dr. Malcolm Newey for his assistance during the year, Matthew Hutchins for setting such high standards to follow and my fellow honours students for being such a great peer group.

Contents

Introduction 1

1 Introducing Standard ML 3
- 1.1 Core Constructs . . . 3
- 1.2 Modules . . . 5

2 The Definition of Standard ML 7
- 2.1 The Core, Modules and Programs . . . 7
- 2.2 The Core Dynamic Semantics . . . 8
  - 2.2.1 The Syntax of the Core . . . 9
  - 2.2.2 The Rules of Inference for the Core . . . 9
  - 2.2.3 Inference Trees . . . 11
  - 2.2.4 The Dynamic Basis . . . 11
- 2.3 Reasoning about Standard ML . . . 12
  - 2.3.1 Related Works . . . 12
  - 2.3.2 The Pattern Matching Theorem . . . 13
  - 2.3.3 A Determinacy Theorem . . . 14
  - 2.3.4 The Associativity of the Append Function . . . 15

3 HOL 17
- 3.1 Higher Order Logic . . . 17
  - 3.1.1 Types, Terms, Formulae and Sentences . . . 17
  - 3.1.2 Deduction and Theorems . . . 18
  - 3.1.3 Theories and Extension . . . 19
- 3.2 The HOL System . . . 19
  - 3.2.1 Primitives . . . 19
  - 3.2.2 User Interface Facilities . . . 20
  - 3.2.3 Pre-Constructed Theories . . . 21
  - 3.2.4 Styles of Proof, Tactics and Tacticals . . . 21
- 3.3 Using HOL to Embed Programming Logics . . . 22

4 Representing The Syntactic Types 25
- 4.1 Type Definition Mechanisms in HOL88 Version 2.0 . . . 25
  - 4.1.1 By Axioms . . . 26
  - 4.1.2 By Representation . . . 26
  - 4.1.3 By Concrete Specification . . . 26
- 4.2 The Mutual Recursion Problem in the Syntactic Types . . . 28

4.3 Mutual Recursion Packages . . . 30
4.3.1 Using the Russo Package in HOL88 . . . 31
4.4 Defining the Syntax . . . 33
4.5 Parsing, Pretty-Printing and Manipulating . . . 33

5 Representing The Semantic Types **35**
5.1 Finite Partial Functions . . . 35
5.1.1 Previous Representations . . . 35
5.1.2 Requirements of a Theory of Finite Partial Functions . . . 37
5.2 A HOL Theory of Finite Partial Functions . . . 37
5.2.1 Defining the Representation . . . 37
5.2.2 More Definitions . . . 40
5.2.3 Stronger Induction Principles . . . 42
5.3 The Semantic Types and More Mutual Recursion . . . 43
5.3.1 Mutual Recursion through Finite Partial Functions . . . 43
5.3.2 Finding a Solution . . . 45
5.3.3 Defining the Semantic Types (at last) . . . 46

6 Representing the Core Semantics **49**
6.1 Problems with Representing the Inference Rules . . . 49
6.1.1 Some more definitions . . . 50
6.1.2 Incorrect Representations . . . 50
6.2 A Correct Representation . . . 53
6.2.1 Completing the Semantics . . . 53
6.2.2 Defining Inference Trees in HOL . . . 54
6.2.3 Eliminating Free Choice . . . 56
6.3 The Dynamic Basis . . . 57

7 Language Properties and Proof Techniques **61**
7.1 Backward Rewrite Rules . . . 61
7.1.1 A Simple Case . . . 62
7.1.2 A Complicated Case . . . 62
7.2 The Pattern Matching Theorem . . . 63
7.3 Forward Inference Rules . . . 63
7.4 The Determinacy Theorem . . . 64
7.4.1 The Significance of the Theorem . . . 66

8 Symbolic Evaluation **67**
8.1 Basic Symbolic Evaluation . . . 68
8.1.1 Evaluating under Assumptions . . . 68
8.1.2 Implementing the Evaluator . . . 69
8.2 Improvements to the Evaluator . . . 70
8.2.1 Evaluating Variables over Complex Environments . . . 70
8.2.2 More Improvements . . . 71
8.3 An Evaluation Tactic . . . 72

9 Reasoning with Standard ML Types **75**
9.1 Concrete Types as Subsets of *Val* 75
9.2 Technique 1: Mapping to an Isomorphic HOL Type 76
9.2.1 Automatic Definition and Theorem Lifting 77
9.3 Technique 2: General Type Definitions 78
9.3.1 Towards Automation 81

10 Verifying Standard ML Programs and Beyond **83**
10.1 The Append Function 83
10.1.1 Stating Correctness 83
10.1.2 The Proof in HOL 84
10.2 A Small Interpreter 86
10.2.1 The Language and its Semantics 86
10.2.2 An Interpreter for the Language 86
10.2.3 Stating Correctness 87
10.2.4 Representing the Types in HOL 87
10.2.5 The Proof in HOL 88
10.3 Computation Properties 89
10.4 The Associativity of `append` 90

11 Conclusions **93**

A Types Needed by the Standard Definition **97**

B Proofs **99**
B.1 The Backward Rewrite Rules 99
B.2 The Pattern Matching Theorem 105
B.3 The Forward Inference Rules 108
B.4 The Determinacy Theorem 109

C Using HOL **113**

D The Finite Partial Functions Theory **117**

E Extracts from the HOL Theories of the Formal Definition **123**

F The HOL Theories of a Standard ML Type **125**

Bibliography **127**

List of Figures

2.1 The Syntactic Breakdown of "`val fst = fn (a,b) => a`" 9
2.2 Inference Tree for "`val x = y`" 12

4.1 The ancestry of the syntactic and semantic object theories 26
4.2 Constructing Types by Isomorphic Mapping from an Existing Type 27
4.3 A Syntax fragment displaying mutual recursion 29
4.4 Our Example Grammar - A Partial Grammar for SML Expressions 30
4.5 Primitive Recursion Theorem for the example grammar 31
4.6 Existence theorem for the example grammar 31
4.7 Induction theorems for the example grammar 32

5.1 A Value displaying mutual recursion through a finite partial function 44
5.2 Definitions of the types *Val* and *Record* 45

6.1 The ancestries of the semantic representation theories 49
6.2 The Implicative Form of the ValBind Axiom 51
6.3 The Equational Form of the ValBind Rule 52

9.1 Representing Standard ML Types via an isomorphic HOL type 77

B.1 The Induction Theorem for Patterns 106

Introduction

Formal reasoning is essentially the process of deducing facts from a rigorously defined theoretical basis. Although formal reasoning is commonly equated with mathematics, scientists from all fields often make use of formal models of their domains. These formal descriptions must be symbolic, mainly for the sake of brevity. They must also be abstract, for they are never a substitute for the real objects which they model.

There is some scepticism in the computer industry regarding how useful it is to study and develop formal reasoning techniques for computer programming languages. Undoubtedly there has been little direct industrial application for these techniques so far. However, what has been learned about languages while studying them formally has contributed significantly to the development of better means of specifying computation. As with physics, which co-exists peacefully with mathematics, it would seem useful to ground common practice in proper formalism.

Programming languages differ from normal objects of mathematical study in several important ways. The complexity involved in describing a computing language is often greater than with numerical systems. However, the subtlety of their structure is not as great. In mathematics it is often possible to reason in terms of abstract categories such as groups and rings, however it is difficult to reason about programs in this way. We usually want to prove facts which are specific to a particular program. The fundamental problem is that the logical structure of most programs is extremely complex, and tends to change rapidly as the program is modified. In an attempt to solve these and other problems, computer scientists have turned to computers themselves to help mechanize the reasoning process.

This project is concerned with aspects of the mechanization of formal reasoning over the programming language Standard ML. Chapter 1 gives a brief introduction to this language. The results of this project shall be based directly on the Formal Definition of Standard ML, and thus in Chapter 2 this definition shall be described. Some examples of how it can be used to reason about the language are also given there. Chapter 3 introduces HOL, the interactive theorem proving system into which the formal definition shall be embedded. Chapters 4 and 5 describe the first two, rather lengthy, stages in this process, where HOL representations for the syntactic and semantic types used in the formal definition are constructed. The representation process is concluded in Chapter 6, where the semantics defined in the Formal Definition are embedded into HOL. An accurate representation of the semantics is, of course, essential. Previous representations have been incomplete at several key points, and a substantially different approach to this problem based on inference trees shall be used.

Chapter 7 describes several important language properties which are central in demonstrating the correctness of the representations developed in Chapters 5 and 6. In that chapter several deriveds sets of theorems which are functionally more useful than the fundamental representation are also derived. Chapter 8 describes a symbolic evaluator which utilises the results of Chapter 7. The work culminates in Chapters 9 and 10 where some important techniques for reasoning with Standard ML types are presented, as are some examples of how the theory can be applied to program verification. A short discussion of the merits and weaknesses of the different tools used in the project is given in Chapter 11.

Chapter 1

Introducing Standard ML

The programming language now known as "Standard ML" has been evolving steadily for the last 20 years. The language was originally developed as the meta-language[1] for the LCF proof system. In [15] Milner describes how the needs of LCF led directly to the main features of the ML language: its functional nature; polymorphism; strong typing; exceptions; and imperative constructs.

LCF eventually led to systems such as HOL. ML, however, continued to attract interest in its own right, accompanied by a general growth in interest in functional programming. In 1985, attempts to produce a definitive version of the language bore fruit with the definition of Standard ML. Standard ML introduced an innovative parameterized module system and greatly refined the core language.

One feature of Standard ML that makes it of special interest to the computer science community is the fact that it has a *formal definition* covering all of the different aspects of the language. There is no other formally defined languaged which comes close to the sophistication of Standard ML. It is expected that all future modifications to Standard ML will be accompanied by appropriate changes and extensions to the formal definition. Of course this means that only extensions which can be rigorously defined will be considered by the ML community. The definition will be the subject of further discussion in Chapter 2.

In this thesis we shall content ourselves with a brief overview of the main features of the Standard ML language. For more detailed information about the language, we direct the reader to the formal definition of the language, the commentary on the formal definition, and the textbooks written on the language [12, 20, 17, 19, 18].

1.1 Core Constructs

Standard ML is first and foremost a *functional* language. Functions are treated as first-class objects and may be used like other values. The following transcript from a Standard ML session illustrates this:

[1] A meta-language is the means we use to speak about a logic. English is the meta-language we usually use to talk about formal systems.

1

```
-  fun rev nil = nil |
=      rev (hd :: tl) = append (rev tl) [hd];
val rev: fn : 'a list  -> 'a list

-  val do_nothing = rev o rev;
val  do_nothing = fn : 'a list  -> 'a list
-
```

We firstly define the function `rev`, which, when applied to a list, evaluates to the reverse of that list. The second declaration shows how freely we may manipulate the function-value we have just defined. The operator "o" indicates composition, and is itself a function which takes two functions as arguments. Thus `rev o rev` evaluates to the composition of the `rev` function with itself. One result of the session is to bind the identifier `do_nothing` to this value.

The session also demonstrates other features of the language. Firstly, it is *strongly typed*. This means that every properly constructed language phrase has associated with it a unique type. Strong typing is generally recognised as essential for serious program development, for it means a large number of errors are eliminated before runtime. Strong typing would be unworkable if every expression had to be subscripted with its type, and thus Standard ML performs *type inferencing* for the user. In addition, the type system is *polymorphic*. In the above example, the symbol `'a` is a type variable and represents an arbitrary type. The system has inferred that the type of `do_nothing` is `'a list -> 'a list` - a function taking a list of elements of some type *'a* to a list of elements of the same type. Polymorphism is essential to the expressive power of Standard ML and enables it to maintain strong typing while allowing quite general functions to be written.

The session illustrates the *interactive* nature of the language. Declarations and expressions are entered directly into the interpreter, or loaded from files. This greatly speeds system development as debugging and testing is quick and easy. Most implementations come with a compiler as well.

Standard ML has extensive features for defining new polymorphic types. Structural types such as lists, trees, and context-free grammar languages may be introduced using the `datatype` declaration:

```
datatype 'a btree = leaf | node of ('a * 'a btree);
```

This definition introduces a new binary tree type `btree` which has labeled internal nodes. The symbol `'a` is again a type variable representing an arbitrary type. For example our new type may be instantiated to types `int btree` and `(string) btree`. The identifiers `leaf` and `node` are introduced as *type constructors* and may be used as values in the system. Lists, which are predefined in Standard ML, could be introduced by such a datatype declaration. Lists are of such importance in Standard ML that they receive a special syntax - `[1,2,3]` has type `int list`. Standard ML has other features typical of functional programming languages, such as an automatic garbage collector and pattern matching.

All of the above features are found in most functional languages. The language is not, however, purely functional, as it supports updatable references and assignments in a fashion similar to an imperative programming language. These are introduced using the special constructor `ref`. Thus `ref []:int list` is a reference to a memory location containing the empty list. The value at the location may be replaced using the assignment operator `:=`. Standard ML also provides an *exception* mechanism, which allows an expression to flag a computation failure. Generally exceptions are reserved to indicate error conditions. Exceptions are typed and may have information attached.

1.2 Modules

The language described so far has been in existence in essentially the same form for over 15 years. In 1985 a proposal to add *modules* to the language was accepted by the ML community. Unlike object-oriented systems, Standard ML maintains a distinction between types and modules. Modules are essentially fragments of environments (structures) and operations on environments (functors). [19] provides a good introduction to the notion of modules in Standard ML. The purpose of adding modules to the system is to support the abstraction of functionality and types at a level beyond polymorphism in lambda abstractions.

Structures may contain types and values (including functions). Each structure has a *signature*. It is helpful to think of a signature describing a structure in the same way a type describes a value, though the analogy is not complete. For example, the structure:

```
structure INT_MAP = struct
  datatype table = T of int list
  fun lookup x (T (a :: table))
        = if (x = a) then true else lookup x (T table)
  fun add x (T table) = T (x :: table)
end
```

implements a very simple integer storage table, essentially a list. It matches with the signature:

```
signature INT_MAP_SIG = sig
   datatype table = T of int list
   val lookup: int -> table -> bool
   val add   : int -> table -> table
end
```

Signatures may be parameterized, and functors may be defined to operate over instances of signatures. For instance, it would obviously be desirable to use a parameterized type `'a` rather than `int` in the above constructs, thus yielding a polymorphic table object.

This project is not directly concerned with modules, for we are not yet in a position to represent their semantics on HOL. However, it would seem natural to perform a verification of Standard ML programs at the module level, and hence they are relevant in the sense that they could well be the ultimate objects that we would like to verify. An excellent statement of correctness may well be the "correctness" of a structure, stated in terms of a series of theorems about the elements of the structure.

Chapter 2

The Definition of Standard ML

This chapter discusses the Formal Definition of Standard ML as set out by Milner, Tofte and Harper in [20]. This is a necessary preliminary before an attempt can be made to reason about the language using the HOLsystem. This project is primarily concerned with the Core language constructs, as any serious reasoning involving the language must begin with these.

2.1 The Core, Modules and Programs

The Formal Definition defines the whole of Standard ML, from its basic syntax through to the effect of interactive evaluations. For historical and practical reasons the language is described on three levels. The lowest of these is the Standard ML Core, which defines a self-contained functional language of expressions, declarations, exceptions and so on. The Core is a significant and rich language in its own right. Modules, mentioned in Chapter 1 form the next layer of the definition. The reason for including modules as a separate definitional layer is partly historical, as their final formulation was not decided until well after the Core language had become stable. The topmost language level is the Program, which is syntactically quite simple and serves to combine the other language constructs.

Each of the three layers of the definition is itself defined in three stages: *syntax*, *static semantics* and *dynamic semantics*. These parts of the formal definition have a close correspondence to what happens when a Standard ML phrase is entered into an interpreter. Firstly, the phrase is *parsed* to ensure its syntactic correctness, as defined by the syntactic definitions. If the phrase parses correctly it is *elaborated*, as defined by the static semantics. Elaboration is the process of ensuring that the phrase obeys Standard ML's typing rules. In addition other pieces of type information, including new type definitions, are extracted from the phrase for use with the elaboration of future phrases. The results of elaboration are stored in the *static basis* for the Standard ML session. After successful elaboration, a phrase will be *evaluated*. Evaluation may result in variables being bound to values, the state of the system being changed due to side effects, and input and output occurring. Evaluation is defined by the dynamic or "runtime" semantics.

Despite the close association between the definition and the implementation of the language, the definition does not simply specify a series of algorithms for these phrases. Rather, it uses the formal style that seems most suited to describing each particular part of the language.

Of the nine parts of the definition, this project will be mostly concerned with the dynamic semantics of the Core. The reason for this is that our main interest lies with proving facts about the execution of Standard ML phrases. Syntax is a well understood area, and is of no interest except in the context of semantics. The static semantics offer a lot of opportunity for proving interesting facts, but we will be better placed to understand how the static and dynamic semantics relate after investigating the dynamic semantics thoroughly. Lastly, Modules and Programs cannot be considered without a solid theory of the Core language, for they are defined in terms of Core constructs[1].

2.2 The Core Dynamic Semantics

In any attempt to describe the semantics of a programming language succinctly and cleanly, a consistent method or style needs to be employed throughout. Several such methods have gained prominence. One of the earliest was Axiomatic Semantics, developed by Hoare, where a language was described using a series of logical axioms, each of which relates the pre and post conditions of a program fragment. Another prominent style is Denotational Semantics, developed initially by Scott and Strachey [7]. Here the aim is to describe the effect of the evaluation of program fragments on objects such as machine memory and environments. Denotational Semantics inevitably requires extensive use of higher-order functions, and perhaps a special purpose logic such as LCF.

The Formal Definition of Standard ML uses a style known as Natural Semantics. An excellent introduction to Natural Semantics and other semantic styles is given in Milner's series of lecture notes [14].

The heart of a definition written in Natural Semantics is a collection of *inference rules*. Each inference rule allows sentences of the form

$$A \vdash \mathit{phrase} \Rightarrow A'$$

to be deduced, given certain premises. Here A forms the *background* to the sentence, and A' the *result*.[2] The interpretation of such a sentence is that against the background A, the phrase *phrase* evaluates to the result A'. The different backgrounds and results allowed in sentences are the *semantic objects* of the definition.

An inference rule is of the form

$$\frac{A_1 \vdash \mathit{phrase}_1 \Rightarrow A'_1 \ \cdots \ A_n \vdash \mathit{phrase}_n \Rightarrow A'_n \ + \text{side conditions}}{A \vdash \mathit{phrase} \Rightarrow A'}$$

Here the top line contains the *premises* to the rule, each of which is either another sentence or a logical side condition. The bottom line is the *conclusion* to the rule.

Natural Semantics is not a complete formal logic in the sense that it requires results from type theory, mathematics and logic to specify the side conditions, phrases, backgrounds and results.

[1]The static semantics of the modules are seen by Milner, Tofte and Harper as their most significant contribution to the field. It is unfortunate that we cannot consider them in detail in this project

[2]Note that the $\vdash$ and $\Rightarrow$ do not have their usual logical meanings in this context, but are used as syntactic sugar for the sentence.

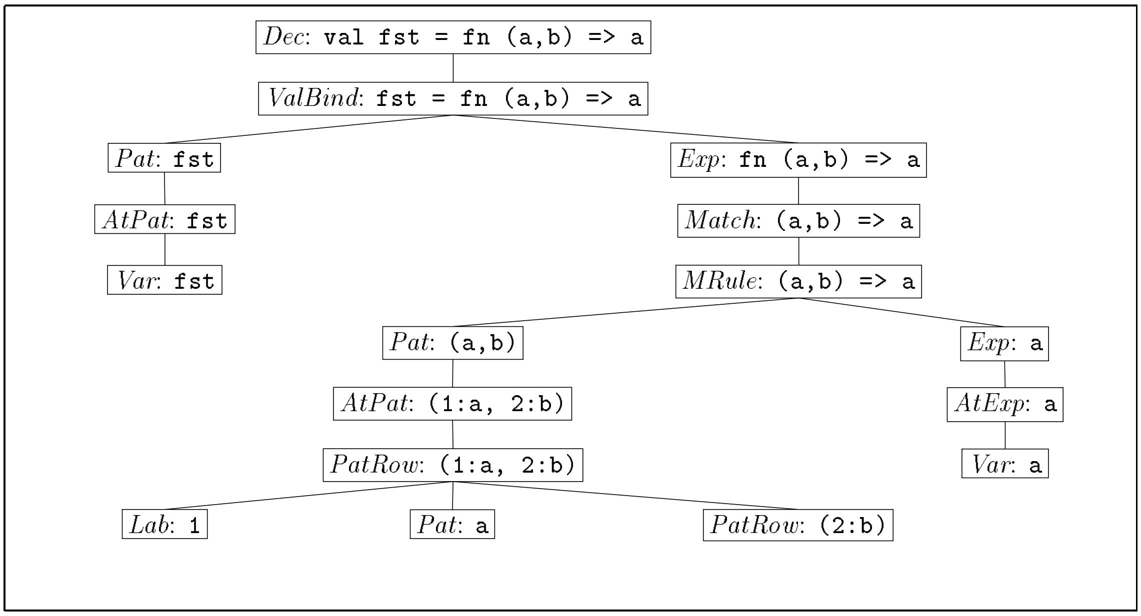

Figure 2.1: The Syntactic Breakdown of "`val fst = fn (a,b) => a`"

2.2.1 The Syntax of the Core

The language used for the Core dynamic semantics is a minimal version of the complete Standard ML Core. Several Standard ML constructs are redundant, in the sense that they may be rewritten in terms of a subset of the language. In addition, the type information of the language is not strictly required for the dynamic semantics, and so all syntactic constructs related to typing are dropped. This leaves the language shown in Appendix A. Eleven different types of syntactic fragments can be used as phrases in sentences: expressions, atomic expressions, expression rows, matches, match rules, declarations, value bindings, exception bindings, atomic patterns, pattern rows and patterns.

The grammatical breakdown of the declaration of the `fst` function:

```
val fst = fn (a,b) => a;
```

is shown in Figure 2.1. This should aid the reader in understanding how most of the syntactic constructs relate.

2.2.2 The Rules of Inference for the Core

There are 57 rules of inference in the dynamic semantics of the Core, each of which deals with a particular case of evaluation. Perhaps the best way to understand the semantics is by example. Rule 104 deals with the evaluation of variables in an environment:

$$\frac{E(var) = v}{s, E \vdash var \Rightarrow v, s}$$

Here E represents an *environment*, which is principally a mapping from variable names to variable values. The rule effectively states

States	s	map addresses to values for referential objects
Variable Environments	VE	map variables to values, including variables to addresses of referential objects
Exception Environments	EE	map exception constructors to exception names
Environments	E	contain the current variable and exception environments
Records	r	map labels to values
Values	v	result from a successful evaluation of an expression
Packs	p	result from an evaluation of an expression which causes an exception

Table 2.1: The major semantic object types

> In the background consisting of state s and environment E, the phrase *var* may be evaluated to the value v, on the condition that the variable *var* maps to v in E. No change occurs to the state.

Both E and s are *semantic objects*. The major semantic objects are described in Table 2.1.

Values, being the results of expressions, are particularly important semantic objects. They include constants, closures, records, constructed values and addresses. Combinations of semantic objects make up the backgrounds and results for the semantic sentences. The combination of objects allowed as a background depends on the particular type of phrase being evaluated, as does the possible result of the sentence. The allowed forms are shown in Table 2.2.

Each form indicates something of the nature of the particular phrase type and its role in the semantics. For example, patterns are evaluated in the background of a state, an environment and a value, and result in a variable environment or FAIL. When a pattern is evaluated it is matched against the value given in the background. If the match succeeds, each variable in the pattern gets associated with a value by the matching process, and thus a new variable environment is produced. The value FAIL is a special value used for a pattern match which fails. Pattern matching shall be discussed further in Section 2.3.2.

There are two conventions employed in writing the inference rules. Firstly, states are left off unless explicitly referred to. When not included, the state information is deduced from the sequencing of the evaluations in the premises of the rule. Secondly, the *exception convention* provides for exception generating sub-evaluations. For each rule whose form, excluding side conditions, is

$$\frac{s_1, A_1 \vdash \mathit{phrase}_1 \Rightarrow A'_1, s'_1 \quad \cdots \quad s_n, A_n \vdash \mathit{phrase}_n \Rightarrow A'_n, s'_n}{s, A \vdash \mathit{phrase} \Rightarrow A', s'}$$

and for each k, $1 \leq k \leq n$, a rule is added of the form

$$\frac{s_1, A_1 \vdash \mathit{phrase}_1 \Rightarrow A'_1, s'_1 \quad \cdots \quad s_k, A_k \vdash \mathit{phrase}_k \Rightarrow p', s'}{s, A \vdash \mathit{phrase} \Rightarrow p', s'}$$

Atomic Expressions	$s, E \vdash atexp \Rightarrow v/p, s'$
Expressions	$s, E \vdash exp \Rightarrow v/p, s'$
Expression Rows	$s, E \vdash exprow \Rightarrow r/p, s'$
Matches	$s, E, v \vdash match \Rightarrow v'/p/\mathsf{FAIL}, s'$
Match Rules	$s, E, v \vdash mrule \Rightarrow v'/p/\mathsf{FAIL}, s'$
Declarations	$s, E \vdash dec \Rightarrow E'/p, s'$
Value Bindings	$s, E \vdash valbind \Rightarrow VE/p, s'$
Exception Bindings	$s, E \vdash exbind \Rightarrow EE/p, s'$
Atomic Patterns	$s, E, v \vdash atpat \Rightarrow VE/\mathsf{FAIL}, s'$
Patterns	$s, E, v \vdash pat \Rightarrow VE/\mathsf{FAIL}, s'$
Pattern Rows	$s, E, r \vdash patrow \Rightarrow VE/\mathsf{FAIL}, s'$

Table 2.2: Allowable sentence forms for different *Phrase* categories

This indicates that if the kth evaluation in the premises results in an exception, then the evaluation in the conclusion results in an exception also. No evaluations beyond the kth are demanded in the premises, corresponding to the fact that exceptions break the natural path of computation. These conventions greatly simplify the rules and together capture the sequential nature of evaluation.

2.2.3 Inference Trees

When fully expanded, there are over 150 inference rules in the Core Semantics. Of these, 20 are base cases similar to rule 104 where there are no sentences among the premises, only side conditions. All evaluations reduce to these primitive base cases.

Successive applications of the inference rules may be thought of as constructing an *inference* tree or *evaluation tree* for a sentence. The children of each node in an inference tree are the sub-evaluations that contribute to the evaluation of the phrase at the node. Consider the evaluation of the declaration

```
val x = y
```

where `y` is bound to the value 1 in the background environment E. The result of this declaration is the binding $\{x \mapsto 1\}$, which is added to E at the program level. This is written as:

$$s, E \vdash \texttt{val x = y} \Rightarrow \{x \mapsto 1\}, s$$

The inference tree for this evaluation is shown in Figure 2.2. Inference trees will play an important part when the formal semantics are embedded into HOL in Chapter 6.

2.2.4 The Dynamic Basis

The inference rules define the evaluation process for the Core language. However, in a Standard ML system there is always a special set of *base values* which are built into the system itself. The base values include numeric, boolean, string, equality, inequality and input/output primitives. The semantics of these values are defined in Appendix D of the Formal Definition in terms of the special function APPLY. This function takes a base value

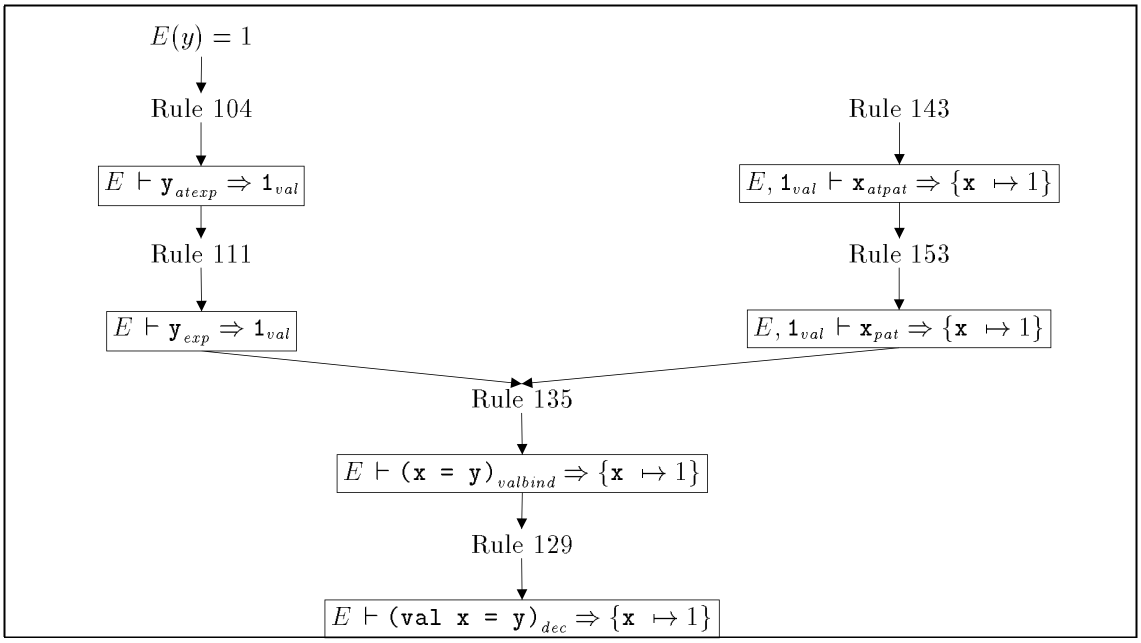

Figure 2.2: Inference Tree for `"val x = y"`

and a value and returns either a value or an exception packet. The base values are all functions, and they are associated with the inference rules by Rule 116:

$$\frac{E \vdash exp \Rightarrow b \quad E \vdash atexp \Rightarrow v \quad \mathsf{APPLY}(b, v) = v'}{E \vdash exp\ atexp \Rightarrow v'}$$

The *dynamic basis* is the initial environment of the Standard ML system. In it, the identifiers used to access the base values are mapped to the base values themselves. Thus the syntactic identifier “+” is mapped to the semantic base value “+”.

2.3 Reasoning about Standard ML

We now turn our attention to the process of using the semantics defined in the Formal Definition to rigorously reason about Standard ML. Firstly we will consider other works directly related to this field. We shall then demonstrate the reasoning process by describing several theorems which are deducible from the definition, giving an outline of their proofs. It is ultimately our aim to prove these and other theorems using HOL.

2.3.1 Related Works

The largest example known to the author of formal reasoning with Standard ML is the verification proof carried out by Aargaard and Lesser [1]. The program they verified implemented an algorithm which automatically designed circuits for logic synthesis. The program was written in a purely functional subset of Standard ML. The verification was *not* based on the Formal Definition of the language, but instead the ML functions were translated directly into functions in the Nuprl proof development system [6]. Thus the verification exercise was reduced to reasoning about mathematical functions directly. This is, of course, one of the beauties of Standard ML, in that its style of definition is so close to mathematics that given enough restrictions we may treat the functions we define as mathematical descriptions.

This project, however, is concerned with reasoning about the full range of the Standard ML Core language constructs, including all its features. Furthermore, we aim to base our reasoning on the Formal Definition, rather than our own interpretation of the language.

A Standard ML verification environment is being developed by Tony Cant at Adelaide, using the Isabelle proof system [4]. The system presently deals with a pure–functional subset of the Core language, and the semantics are embedded in Isabelle's first-order logic. It would seem that the representation they use may well suffer from some of the incompleteness problems that will be encountered in Chapters 4 and 6.

Two good sources of examples of reasoning based on the formal definition are available to us. The first is the project by Matthew Hutchins, which forms the immediate background to this thesis. In chapter 2 of [12], Hutchins provides a detailed proof of the non-trivial append associativity theorem. This is not a verification theorem, as it asserts the semantic equivalence of two program fragments. This theorem shall be one of the main examples of this thesis, and shall be discussed shortly.

Another source of examples is the Commentary on the Formal Definition, written by Milner and Tofte ([17]). In their chapter on the Core dynamic semantics they discuss several important results about the language. These shall be considered in the following sections.

2.3.2 The Pattern Matching Theorem

Pattern matching lies at the heart of function application in Standard ML. The process is best described by example. Consider the definition of a function which selects the last two elements of a quadtuple:

```
fun last_two (_,_,x,y) = (x,y);
```

"`(_,_,x,y)`" is a pattern, where the `_` characters are wildcards. When the function is applied to a value, the value will be matched with the pattern, producing new bindings for x and y. These are then used when evaluating the expression on the right of the function definition.

The interesting fact about pattern matching is that it has no side-effects, as no expressions are actually evaluated. However, this fact is not stated explicitly in the formal definition - it must be proved. The Commentary on Standard ML states this property in the following way:

Theorem 1 (Pattern Matching) *Let E, v and pat be any environment, value and pattern. Suppose that*

$$s, E, v \vdash pat \Rightarrow r, s'$$

can be inferred for states s, s' and result r. Then r is either a variable environment VE or else the special value FAIL *- it cannot be an exception packet p. Moreover, $s = s'$.*

The proof given in the commentary is straight forward. It is conducted by structural induction over the possible values for *pat*. Consider a step case, say when the pattern is a part of a pattern row *pat*, *patrow*. The relevant rule is Rule 152, expanded by the state convention:

$$\frac{s_0, E, r(lab) \vdash pat \Rightarrow VE, s_1 \qquad s_1, E, r \vdash patrow \Rightarrow VE'/\mathsf{FAIL}, s_2}{s_0, E, r \vdash lab\ \texttt{=}\ pat\texttt{,}\ patrow \Rightarrow VE + VE'/\mathsf{FAIL}, s_2}$$

With the induction assumption that no changes in state occur in the sub-evaluations, it can be seen that no change in state occurs for the inferred sentence. Similarly it can be seen that the rule only allows variable environments or FAIL as the value for the inferred sentence. A similar result holds for all the sub-cases of the induction, giving us the required theorem. We shall prove this theorem in HOL in Section 7.2.

2.3.3 A Determinacy Theorem

The second theorem we consider from the Commentary concerns the deterministic nature of evaluation. It is not at all obvious from the rules that the evaluation process is in fact deterministic. This is a fundamental property that we require of a language - that if we run the same code twice we get the same result. The theorem can be stated as follows:

Proposition 1 (Determinacy) *Let the two sentences*

$$s, A \vdash phrase \Rightarrow A', s' \quad s, A \vdash phrase \Rightarrow A'', s''$$

both be inferred. Then $A' = A''$ *and* $s' = s''$*.*

Unfortunately, this result does not strictly hold with Standard ML. We see why if we consider Rule 114, where a new memory location is added to the state upon evaluation of a `ref` application:

$$\frac{s, E \vdash exp \Rightarrow \texttt{ref}, s' \quad s', E \vdash atexp \Rightarrow v, s'' \quad a \notin \mathrm{Dom}(mem\ \mathrm{of}\ s'')}{s, E \vdash exp\ atexp \Rightarrow a, s'' + \{a \mapsto v\}}$$

The top line requires the choice of a new, unused memory address a at which to store the referenced value. The exact value to choose is not specified, all we know is the fact that this address must be unused. The same problem applies when exception names are chosen for exception constructors. It is these two rules which make the semantics essentially non-deterministic.

Harper and Milner modify the statement of their determinacy theorem as follows:

Theorem 2 (Determinacy - Harper and Milner) *Let the two sentences*

$$s, A \vdash phrase \Rightarrow A', s' \quad s, A \vdash phrase \Rightarrow A'', s''$$

both be inferred. Then (A'', s'') *only differs from* (A', s') *by a one-to-one change of addresses and exception names which do not occur in* (s, A)*.*

The proof of this theorem is by computational induction on the length of the inference process. For any particular inference step, it can be shown that there is only one path that the inference process could have taken. For rules such as 114, it can be shown that the one-to-one change of address property is not disturbed.

We shall reserve the details of the proof to Section 7.4, where the theorem shall proven in HOL. By introducing arbitrary choice functions into the semantics, we will also eliminate the highly undesirable side condition present in the theorem above.

2.3.4 The Associativity of the Append Function

The third theorem about Standard ML that we shall describe is a property of the `append` function. The declaration of the function is shown below. The function has type `('a list * 'a list) -> 'a list`.

```
fun append (nil, M)      = M |
    append (cons x L, M) = cons x (append (L,M));
```

The property of the function that we wish to prove is its associativity, loosely stated as the equivalence of two program fragments:

$$\texttt{append}(A, \texttt{append}(B, C)) \equiv \texttt{append}(\texttt{append}(A, B), C) \tag{2.1}$$

It can be seen that this statement is incomplete in several important senses. Firstly, we must define what is meant by the equivalence of two program fragments. Certainly we are not asserting the syntactic equality of the two phrases. However, what is implied is equality over inferability. That is, two expressions *exp* and *exp'* are equivalent if, for every state s, environment E, result res and new state s'

$$s, E \vdash exp \Rightarrow res, s' \quad \text{iff} \quad s, E \vdash exp' \Rightarrow res, s'$$

The next problem with statement 2.1 involves the typing of the expressions. What we need to know is exactly what information can be assumed about the type of value that A, B and C produce when they are evaluated. In [12] Hutchins demonstrates that the associativity result does, in fact, hold for arbitrary expressions A, B and C regardless of the values that might be inferred for them. This result is based on the fact that badly typed expressions will cause a `[Match]` exception when all the pattern matches fail in the application of the `append` function. This result, although interesting, is not exactly what we want to prove. For instance, one of the things that we might want to prove about the `append` function is that it will *never* cause an exception of any kind. This result can only be shown with extra information about the types of A, B and C.

A full theory of typing and evaluation would involve both the static semantics and dynamic semantics of the Core language. A proof would be necessary of the relationship between the two - i.e. that the dynamic semantics preserve the type structures developed in the static semantics. This property is beyond the scope of this thesis and is perhaps over-complicating the issue anyway. The simplest solution is to characterize the type information for the expression A by assuming that, if we may infer a value *val* for A which is not an exception, then *val* has the structure of a list. That is, it is either the value `nil` or a constructed value `cons` *hd tl*. This information is sufficient to characterise the expected types.

It is important to consider the effect that state changes and exceptions might have on the truth of 2.1. It is plausible to interpret the statement in two ways - either saying that no state changes occur (and thus demanding that A, B and C be pure functional expressions - a very weak result but much easier to prove), or that both sides of the equation cause the same state change. Note here that the semantics of Standard ML ensure that the evaluation order remains the same regardless of the placing of the parentheses.

Lastly, implicit from 2.1 is the requirement that the identifier `append` be mapped to the correct value in the environment in which the evaluation is taking place. The value is a closure determined by the semantics of the declaration of the function.

In light of these complications, the following is a more appropriate statement of the associativity property:

Theorem 3 (Append Associativity) *Given arbitrary expressions A, B and C, states s_0, s_1, s_2, s_3, environment E and the following assumptions:*

$$
\begin{array}{lccc}
 & \multicolumn{3}{c}{E(\texttt{append}) = \textit{the closure generated by the declaration of } \texttt{append}} \\
\textit{and} & s_0, E \vdash A \Rightarrow list_A, s_1 & \textit{or} & s_0, E \vdash A \Rightarrow pack_A, s_1 \\
\textit{and} & s_1, E \vdash B \Rightarrow list_B, s_2 & \textit{or} & s_1, E \vdash B \Rightarrow pack_B, s_2 \\
\textit{and} & s_2, E \vdash C \Rightarrow list_C, s_3 & \textit{or} & s_2, E \vdash C \Rightarrow pack_C, s_3
\end{array}
$$

then

$$
\begin{array}{l}
s_0, E \vdash \texttt{append (}A\texttt{,append (}B\texttt{,}C\texttt{))} \Rightarrow res, s' \quad \textit{iff} \\
s_0, E \vdash \texttt{append (append (}A\texttt{,}B\texttt{),}C\texttt{)} \Rightarrow res, s'
\end{array}
$$

Not all the assumptions are needed all the time. For instance, if the evaluation of A causes an exception, then no assumption need be made about the evaluation of B or C. If we combine this result with the determinacy theorem of the previous section, then the equivalence of our program fragments is ensured.

The proof of the property is fairly simple for the cases where exceptions occur in the evaluation of A, B or C, for the exceptions just get propagated in either code fragment. This leaves us with the case when all three of A, B and C evaluate to lists. The proof then follows by verifying the correctness of the `append` function, in the sense that we prove that the function always does indeed evaluate to the appending of its two arguments, if its arguments are lists[3]. Once we have proven this property, we may ignore the Standard ML code and merely prove that appending list values is associative.

The full proof of the property is carried out in HOL in Chapter 10. There it is also shown that the `append` function will never cause an exception, will never cause a change of state and will always terminate.

[3] This, of course, means that we have to define the appending over list values

Chapter 3

HOL

In this chapter we introduce HOL - the reasoning tool which will serve as our platform for constructing a formal theory of SML. In the introductory note accompanying HOL, the first thing that the principal author of the system Mike Gordon writes is:

> The HOL system is not user friendly...In the hands of an expert HOL is a powerful and robust tool; in the hands of an untrained novice it may lead to frustration, disillusionment and despair...

With this two-edged caveat in mind, we shall proceed to describe aspects of the logic that forms the basis of the HOL system. We shall then examine how the logic is implemented as a computerised reasoning system.

There are presently several implementations of the HOL system. The one which was used for this project was HOL88 Version 2.0, developed at the University of Cambridge. Of future importance will be HOL90, which is built using Standard ML as its meta language, rather then the early version of ML found in HOL88 [3]. ICL Secure Systems use a HOL derivative in their commercial work known as Proof Power.

3.1 Higher Order Logic

To the newcomer, the logic of HOL often seems somewhat confusing. On the one hand, it contains many familiar constructs such as $\forall$ and $\exists$ quantification and others found in predicate calculus. However, the actual theoretical formulation of the logic is a rather esoteric theory of types and choice operators which seems somewhat unrelated to the uses of the system. It is, however, one of the strong features of the system that it can be understood on several levels, so the beginner need not understand all the subtleties of the underlying logic before using it.

3.1.1 Types, Terms, Formulae and Sentences

The HOL logic is based on Church's simple theory of types [5], extended to include type variables as part of the logic itself. There are four sorts of terms in the logic: constants, variables, function applications and λ-abstractions. Types, written as subscripts, are assigned to each term, and terms must obey polymorphic typing rules not unlike those of ML. Types in HOL always contain at least one member.

The logical constructs of HOL are built from the type *bool* and the operators $\supset_{bool\to(bool\to bool)}$, $=_{\alpha\to(\alpha\to bool)}$ and $\varepsilon_{(\alpha\to bool)\to\alpha}$. These represent implication, equality and choice respectively. From the constant specification rule of the logic, the following logical operators and constants are defined:

$\mathbf{T}_{bool}$	truth
$\mathbf{F}_{bool}$	falsity
$\neg_{bool\to bool}$	negation
$\wedge_{bool\to(bool\to bool)}$	conjunction
$\vee_{bool\to(bool\to bool)}$	disjunction
$\forall_{(\alpha\to bool)\to bool}$	universal quantification
$\exists_{(\alpha\to bool)\to bool}$	existential quantification
$\mathrm{COND}_{bool\to(\alpha\to(\alpha\to\alpha))}$	conditional

Formulae are the basic statements to which we assign truth or falsity, and are thus characterised as *bool* typed terms. *Sentences* consist of a finite set of assumptions and a conclusion, all of which are formulae.

The ε-choice operator corresponds to free (yet deterministic) choice over a set, which in turn corresponds to the mathematical Axiom Of Choice. While often useful in definitions, it is a difficult operator to use in mechanical proofs - for an excellent description of the semantics of the operator and techniques available to eliminate it from proofs, see [10].

3.1.2 Deduction and Theorems

One of the first things a newcomer to HOL notices is the restrictions on the production of *theorems*. Theorems in HOL correspond to sentences which are logically deducible from a set of axioms using a set of primitive *rules of inference*. Theorems may not be created arbitrarily - the logical validity of a sentence must be demonstrated before it will be admitted to theoremhood. Theorems are denoted by the use of the turnstile symbol, with the assumptions (Γ) of the theorem on the left:

$$\Gamma \vdash t_1$$

The production of theorems, collected together into a *theory*, is the ultimate aim of any HOL session.

Inference rules are written in the traditional form:

$$\frac{\Gamma_1 \vdash t_1 \quad \cdots \quad \Gamma_n \vdash t_n}{\Gamma \vdash t}$$

where the hypotheses of the rule are written above the line, and the conclusion of the rule written below the line. There are eight primitive rules of inference in the HOL system, each of which is documented in [9]. They include the rule of modus ponens (`MP`), a rule of substitution (`SUBST`), a rule to eliminate type variables (`INST_TYPE`) plus rules to introduce assumptions, equality and lambda abstractions.

The HOL system incorporates five basic axioms, defining the semantics of implication, lambda abstraction, choice and the *bool* type. They also assert the existence of an infinite type *ind* from which other types can be derived.

As with the syntax of the HOL logic, there is little need for the beginner to understand either the basic inference rules or axioms of the system. It is far more helpful to reason from an existing set of pre-proved theorems, using *derived* inference rules. These are composed in terms of the basic inference rules using the ML meta-language.

3.1.3 Theories and Extension

It can be seen that the HOL logic described so far allows the deduction of arbitrarily complex theorems in terms of the type domains *bool* and *ind*. In effect, a *theory* has been constructed, containing two types, a set of constants and a set of deducible theorems. This concept is integral to HOL. The result, if any, of a session with HOL is a theory about the constants and types which have been specified. In the implementation of HOL a subset of the deduced theorems are saved along with the theory. A HOL theory is thus a set of definitions and axioms, plus a finite set theorems which can be deduced from that basis.

Theories constructed in HOL are seen as *extensions* to the initial theory. Theories may be extended by two primitive rules:

- Extension by constant specification
- Extension by type specification

In the logical sense, theories may *not* be extended by adding new axioms. Extension is a process which preserves the logical validity of the initial theory, and there is no guarantee that a new axiom will not introduce contradictions into the system[1]. Only some special cases of type specification are implemented in HOL88 Version 2.0. We will return to the type specification primitives in Chapter 4.

3.2 The HOL System

3.2.1 Primitives

When fully constructed, the HOL system contains an initially overwhelming collection of theories, types, constants, derived inference rules, derived alternative reasoning strategies, derived specification methods and implementation features not directly associated with the logic of the system. It is with these rather than the underlying logic that this project will be most concerned. However, it is useful to consider the implementation of the core logic, as its style is duplicated throughout the derived parts of the system.

As mentioned previously, an early version of the ML language is used as the *meta language* for HOL88. This means that all the basic constructs of the HOL logic - terms, types, theorems etc. are represented as typed ML values and are manipulated by fragments of ML code. Furthermore, the primitive rules of inference and theory extension are implemented as ML code. Derived rules and functions are also written in ML, being constructed out of the core ML functions.

Part of the core implementation of the HOL logic is a set of functions to manipulate types and terms with respect to their components. For example, the functions

```
mk_imp : (term # term) -> term
dest_imp : term -> (term # term)
is_imp : term -> bool
```

[1] For a formalisation of the notion of "logical validity" in HOL in terms of models and standard models, see [9].

construct, destruct and test for terms of the form $t_1 \supset t_2$. Similar functions are defined to manipulate higher level constructs, for instance the functions `list_mk_forall` and `strip_forall` construct and destruct multiple $\forall$ quantifications from terms. Similar functions may be written for any syntactic structures defined in HOL theories.

Deductions are performed in ML by using functions which implement the eight primitive rules of inference. The function:

```
ABS : term -> thm -> thm
```

implements the basic lambda-abstraction rule of inference. Other primitive rules of inference are defined similarly. There are no core constructors for objects of type `thm`, which is consistent with the requirement that all theorems be justified by a deduction using the rules of inference.

Theory hierarchies are specified in HOL through the use of the primitive functions `new_theory` and `new_parent`. Extensions are made to existing theories by using the functions `new_specification` and `new_type_definition`, which introduce new constants and types into the theory. Although adding axioms does not constitute a logically sound method of extending a theory, it is still allowed through the use of `new_axiom`. Deduced theorems may be added to the set of theorems stored in a theory by using `save_thm`.

In summary, the fundamental logic of the HOL system is implemented through the following classes of core functions:

- functions to manipulate terms and types
- functions to implement the primitive rules of inference
- functions to support the construction of theories and theory extension.

The remainder of the HOL system is built on top of this core.

3.2.2 User Interface Facilities

Some basic interface functionality is provided by HOL:

- Terms can be entered directly through *quotation*. This is implemented by altering the ML parser to recognise a special syntax for terms. Without this device, terms could only be created using the constructor functions, which would be intolerably tedious.
- The quotation parser automatically infers types for many terms in a similar manner to the ML type inference process. In this way, not every term need have a type subscribed.
- Terms and types are presented to the user through a *pretty-printer*. Without pretty printing, it would be effectively impossible to understand the logical meaning of the terms being displayed.

3.2.3 Pre-Constructed Theories

Part of the beauty of formal systems comes from the fact that properly constructed and logically sound theories often enable the construction of further, more complicated theories. For instance, set theory is fundamental to nearly all areas of mathematics. To this end, the HOL logic allows theories of polymorphic types such as sets to be constructed and used in other theories. The HOL system comes complete with several such pre-constructed theories. Some of the theories are only present for historical reasons, but most contain quite useful constructions. The most important theories are those of paired-values, lists, trees and natural numbers. Extensive use of these theories will be made in this project. Each comes with a set of pre-proved theorems concerning these objects. Also available is a library of HOL theories which include sets, bags, finite sets, ascii values, strings, a theory of fixed points, integers and some elementary group theory.

3.2.4 Styles of Proof, Tactics and Tacticals

It has long been recognised that human reasoning is rarely a process of beginning with an existing set of knowledge and working to some logical conclusion. The classical distinction that is made is between *forward* and *backward* proof. In backward proof we begin with the sentence we wish to prove, and by a process of reduction show that the sentence must be true given our existing knowledge.

At the primitive level, the HOL rules of inference support only forward proof. No direct allowance is made for reduction as a valid logical process. However, HOL does provide support for backward proof, implemented as a "derived" proof style based on the primitive functionality. It is this style of proof which forms the most natural introduction for the beginner seeking to understand the HOL system. Even for experienced HOL users at least 90% of theorem deduction is carried out using this technique.

HOL uses two mechanisms to support backward proof. Firstly, a wide range of pre-defined *tactics* and *tacticals* are provided. Tactics were first used in LCF, and intuitively they are "methods" by which to reduce complex goals into simpler sets of subgoals. In HOL a tactic is an ML function of the following type:

```
lettype tactic = goal -> (goal list # (thm list -> thm))
```

The first product of the application of a tactic is the set of subgoals which need to be solved to complete the proof of the original goal. The second is a *proof function*, which, when supplied with a proof to each of the subgoals, will produce a theorem solving the original goal. In HOL a *goal* is a logical sentence, i.e. a set of assumptions and a conclusion.

The second mechanism provided by HOL is the *subgoal package* which supports the interactive backward proof of theorems. To illustrate the use of backward proof, consider proving the statement:

$$\forall x.\exists y.x = y$$

The following is the transcript of a HOL session solving this goal.

```
#g "∀ (x:*). ∃ y. x = y";;    %< place the subgoal on the stack       >%
"∀ x. ∃ y. x = y"

#e GEN_TAC;;                 %< use GEN_TAC to solve for one         >%
"∃ y. x = y"                  %< arbitrary x                          >%

#e (EXISTS_TAC "x:*");;      %< use EXISTS_TAC to suggest that x is  >%
"x = x"                      %< the value we need for y              >%

#e(REWRITE_TAC []);;         %< use REWRITE_TAC to eliminate         >%
goal proved                  %< a trivial tautology                  >%
|- x = x
|- ∃ y. x = y
|- ∀ x. ∃ y. x = y             %< the theorem which we have proved     >%
```

The example illustrates the use of the tactic `REWRITE_TAC`, amongst others. Rewriting plays a central role in HOL in both forward and backward proof, and is essentially the process of replacing terms with logically equivalent, but more useful ones. A range of simple rewrites are used by `REWRITE_TAC`. Without these there would be a constant need to reconstruct the forward proofs for simple tautologies such as `|- x = x` in the example above.

Due to the power of the ML meta language, there is no requirement in HOL that proof be carried out in a strictly forward or backward manner. Though not used in this project, HOL88 contains a library which provides support for "windowed" reasoning, which conforms to neither of these models. Styles of proof are often mixed freely within the proof of a theorem.

3.3 Using HOL to Embed Programming Logics

It is the aim of this project to *embed* the formal definition of Standard ML in HOL. Michael Gordon's paper [8] illustrates the practicality of this approach. His object of study is a simple imperative language for which Hoare's axiomatic semantics hold. Gordon first describes the representation of the syntactic constructs of the language in HOL using the type definition features of the system. Then, after giving a simple denotational description of the language, he proceeds to deduce Hoare's axioms from that basis. He then demonstrates that mixed forms of reasoning, both propositional and denotational, can then be used to reduce the verification of a simple programs to logical side conditions. These can then be proven as lemmas, still within the HOL framework. Furthermore, he proceeds to prove that other forms of reasoning such as Dijkstra's weakest precondition semantics are also valid.

It is difficult to imagine a proof as complete or rigorous as Gordon's being performed in any environment less powerful than the HOL system. A specialized reasoning tool would almost certainly be based on one form of semantics only, with no clear path available to prove the validity of different forms of reasoning. In addition, in any such work we need to define a range of objects particular to the domain over which the program is operating. For instance, a program which sorts a list will need some associated "theory" of lists. Such a theory can be constructed formally with a tool like HOL.

In this project, we shall initially follow a similar course to Gordon. However, the complexity of Standard ML will make our task more difficult, and its functional nature makes for a different approach to program verification.

Chapter 4

Representing The Syntactic Types

In this chapter and the next we come to the first major tasks that face us in using HOL to reason with Standard ML. These involve the construction of representations for the types used in the Formal Definition. Two groups of constructs need to be represented:

1. Syntactic objects representing portions of Standard ML programs.
2. Semantic objects such as environments, states and values.

The semantic objects make use of the syntactic, so naturally the syntactic must be constructed first. One HOL theory is constructed for each group - `SML_syntax` and `SML_semantics`. The ancestry tree of the relevant theories is shown in Figure 4.1, along with the "supporting" theories `fpf`, `string`, `sets`, `HOL` and `ascii`. All of these are built into HOL except `fpf`. The construction of the `fpf` theory will be described in the next chapter.

Many of the problems encountered and described in this section will only be clear if the reader is familiar with the type structures that we need to define in HOL. The full definition of these type structures can be found in Appendix A. We shall consider examples of each of the problems encountered as they arise.

For the purposes of this project several features of the Core language shall be ignored, leaving them to be added to our theories at a later date. In particular, we do not consider input/output, real numbers or strings. It would not be too difficult to add these constructs to our theories. In addition we restrict the range of integers to a range of the natural numbers.

The syntactic and semantic constructs were originally embedded into HOL by Matthew Hutchins, and it is helpful to begin the discussion of our own representation by analyzing some problems that he encountered. However, as the representation process is almost exclusively concerned with defining new types, we shall firstly overview the type definition mechanisms available in HOL88 Version 2.0. This will aid us in our understanding of HOL's type theory and the nature of the theorems that are needed to characterise different types.

4.1 Type Definition Mechanisms in HOL88 Version 2.0

HOL88 2.0 provides three mechanisms by which types may be introduced into the system: by axiom, by representation and by concrete specification.

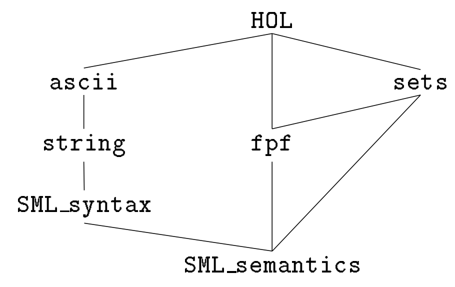

Figure 4.1: The ancestry of the syntactic and semantic object theories

4.1.1 By Axioms

Types may be specified with *no* semantics by using the function `new_type`. The call

```
new_type 0 `foo`;;
```

extends the current theory by adding a new type *foo*. We know nothing about the type apart from the fact that it has at least one element. This mechanism is essentially useless without adding axioms which define the properties of the type in some way.

4.1.2 By Representation

The second mechanism uses the type definition rule of the system directly. The new type is constructed by making a *representative set* out of existing types and setting up a one-to-one mapping between that set and the new type. This is illustrated in Figure 4.2. The method is implemented via the function

```
new_type_definition: (string # term # thm) -> thm
```

which expects an identifying name for the type produced, a boolean valued λ-expression defining the representative set, and a theorem stating that at least one member of the set exists. Using this method is generally lengthy and it is often hard to discover appropriate representation sets.

4.1.3 By Concrete Specification

The third mechanism can only be used on a restricted class of types, and is ultimately implemented in terms of the primitive type specification rule mentioned above. The subset of types supported are those which are both *finite* and *concrete*. Concrete types are inherently "structured" in that each value of the type can essentially be constructed in one and only one way. For example, lists and trees are concrete types, whereas bags, sets and functions are not. By finite we mean that any instance of the type may be constructed out of a finite number of steps. For instance, the natural numbers are finite in this sense,

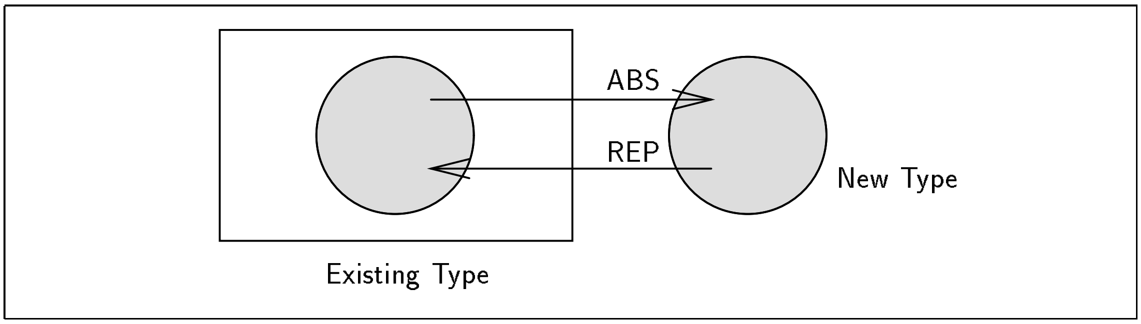

Figure 4.2: Constructing Types by Isomorphic Mapping from an Existing Type

for they are each individually finite. In general, the finite, structured properties of a type may be captured in terms of a *primitive recursion* theorem. This theorem asserts that any specification of a recursive function over the type structure gives rise to a unique corresponding actual function.

When trying to understand the type theory of such structures, the following guide is useful:

1. The *existence* portion of a primitive recursion theorem is sufficient to capture the structural properties of the type.

2. The *uniqueness* portion of a primitive recursion theorem is sufficient to justify an induction principle over the type.

3. An *induction principle* is sufficient to guarantee the finiteness of a type, whether it is structured or unstructured.

4. For very simple structured types such as the natural numbers, an induction principle may be sufficiently powerful to justify primitive recursion.

The interface to the concrete specification mechanism is by the function

```
define_type: string -> string -> thm.
```

The `define_type` mechanism expects as input a specification for the type, which is similar to an EBNF grammar except only one non-terminal type may be specified. The exact syntax allowed is:

$$op = C_1 ty_1^1 \ \ldots \ ty_1^{k_1} \mid \ldots \mid C_m \ ty_m^1 \ ... \ ty_m^{k_m}$$

where each ty_i^j is either a type expression already defined as a type in the current theory, or is the type *op* being defined. The constants C_i are *constructors* which are introduced into the system as functions producing objects of type *op*. The function returns a primitive recursion theorem for the type. A facility is provided to automatically prove the induction principle for this type.

As an example, we may use `define_type` to define a new binary tree type:

```
define_type `BTREE_DEF` `btree = LEAF | NODE * btree btree`;;
```

The primitive recursion theorem produced is:

1

$$
\begin{aligned}
&\texttt{BTREE\_DEF} = \vdash \forall e\ f.\\
&\qquad \exists_{unique} fn.\\
&\qquad (fn\ \texttt{LEAF} = e) \wedge\\
&\qquad (\forall x\ B1\ B2.\ fn(\texttt{NODE}\ x\ B1\ B2) = f\ (fn\ B1)(fn\ B2)\ x\ B1\ B2)
\end{aligned}
$$

The induction principle proved is:

2

$$
\begin{aligned}
&\texttt{BTREE\_INDUCT} =\\
&\quad \vdash \forall P.\\
&\qquad P\ \texttt{LEAF} \wedge\\
&\qquad (\forall left\ right.\ P\ left \wedge P\ right \supset (\forall x.\ P(\texttt{NODE}\ x\ left\ right))) \supset\\
&\qquad (\forall btree.\ P\ btree)
\end{aligned}
$$

LEAF is a polymorphic constant defined along with the new definition and has type *(*)btree*. NODE is a constructor for the type and itself has type $* \rightarrow (*)btree \rightarrow (*)btree \rightarrow (*)btree$.

4.2 The Mutual Recursion Problem in the Syntactic Types

All three type definition mechanisms outlined in the previous section have major limitations. Using `new_type` necessitates the use of `new_axiom` which may introduce logical inconsistency into the system. Using `new_type_definition` directly is difficult and would be an overwhelming task for a type structure as large as that needed for Standard ML. Lastly, we shall see that `define_type` is too limited to use to construct either the syntactic or semantic constructs we need.

We are aiming to define the syntax of the Standard ML Core as a set of types in HOL. There are eleven distinct phrase classes in syntax. A partial syntax tree for the Standard ML code

```
f (g ...)
```

is shown in Figure 4.3. This piece of code demonstrates the fundamental reason why we cannot use HOL's concrete type definition package to define our type structure - *mutual recursion* occurs between the types. The type *Exp* is defined in terms of the type *AtExp*, and *AtExp* in terms of *Exp*. This is not the only instance of this problem - from the grammar of the core (see Appendix A) it can be seen that many other mutual dependencies occur.

Hutchins' solution to the mutually recursive types problem, although faulty, is instructive:

1. All the circularities in the syntactic and semantic type definitions can be traced back to the types *Exp*, *Pat* and *Val*. Thus, he firstly introduced these new types using `new_type`.

2. The remaining types *AtExp*, *ExpRow*, *Match*, *Dec* etc. were then introduced using `define_type`, which merely involved specifying the relevant fragments of the grammar.

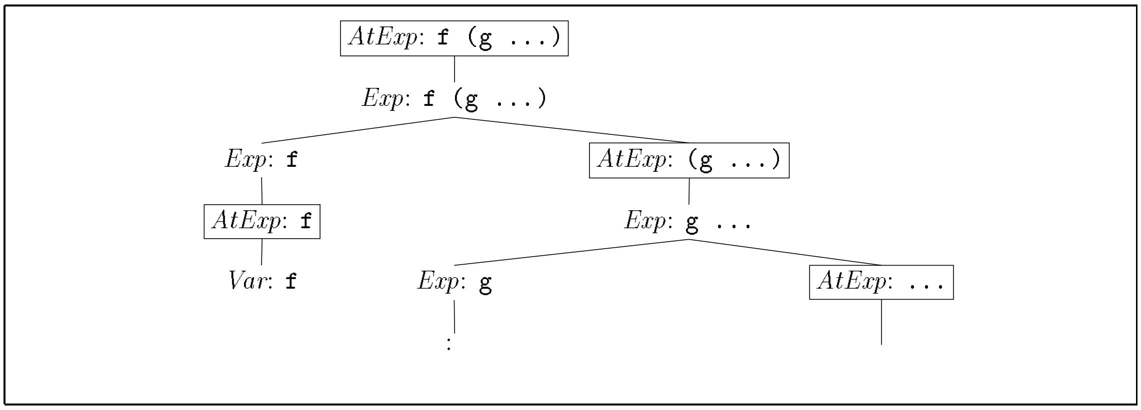

Figure 4.3: A Syntax fragment displaying mutual recursion

3. The troublesome types introduced in step 1 then had their primitive recursion theorem introduced as an axiom of the theory, as if they had been defined using `define_type`.

This solution seems quite reasonable, as we have now asserted something about each type and it seems possible that by using `new_axiom` we have successfully captured the type structure needed. We shall now investigate why this is not so.

One primitive recursion theorem and one induction principle is available for each type. For instance, the induction principle for the *Exp* is:

3

$$
\begin{array}{l}
\vdash \forall P\_Exp. \\
\quad (\forall A.\ P\_Exp(\texttt{Exp\_AtExp}\ A)) \wedge \\
\quad (\forall E\ A.\ P\_Exp\ E \supset\ P\_Exp(\texttt{Exp\_Appl}\ E\ A)) \wedge \\
\quad (\forall E\ M.\ P\_Exp\ E \supset\ P\_Exp(\texttt{Exp\_Handled}\ E\ M)) \wedge \\
\quad (\forall E.\ P\_Exp\ E \supset\ P\_Exp(\texttt{Exp\_Raise}\ E)) \wedge \\
\quad (\forall M.\ P\_Exp(\texttt{Exp\_fn}\ M)) \supset \\
\quad (\forall E.\ P\_Exp\ E)
\end{array}
$$

If the concrete type structure has been properly defined, then it should be possible to prove both a primitive recursion theorem and an induction principle over the *whole* type structure. In the case of the induction theorem, the induction hypotheses should require that a predicate be proven for each of the possible leaf nodes of the concrete type tree, and also for each of the "step cases". To demonstrate that none of the induction theorems available are sufficient, consider the theorem produced for the *Exp* type. Each conjunct in the assumptions indicates an inductive case that must be proved. We can see the difficulty that would be found in trying to prove the first of these:

$$P\_Exp\ (\texttt{Exp\_AtExp}\ A)$$

$AtExp$	$=$	AtExp_Con Con
	$\mid$	AtExp_Paren Exp
Exp	$=$	Exp_AtExp $AtExp$
	$\mid$	Exp_Con Con

Figure 4.4: Our Example Grammar - A Partial Grammar for SML Expressions

No assumption can be made about the atomic expression A, and as atomic expressions are defined in terms of expressions, we will almost certainly need the step-case assumption $P\_Exp E_1$ for some sub-expression E_1 of A. The mutual recursion between the types means we have no well-founded base cases for inductive proofs.

Effectively the finiteness of the types involved has not been captured. The problem is in fact worse than this, for the use of `new_axiom` has introduced almost certainly introduced logical inconsistency into the system by allowing an unfounded type.

4.3 Mutual Recursion Packages

Mutually recursive types have been the subject of discussion between HOL users for the last several years. Tom Melham, one of the principal researchers into the HOL system, has demonstrated how to perform a manual constructions of simple type structures. In addition, several groups have worked on producing packages to automate this process.

Of the packages that have been developed, only one appears to be both correct and useful - that developed during 1992 by Claudio Russo at the University of Edinburgh [21]. It is this package which we will use initially to define our type structures. The package is capable of defining types for mutually recursive grammars of a definitional style similar to those handled by the existing HOL `define_type` package. For instance, the grammar in Figure 4.4, which is part of the Standard ML syntactic grammar defining the *Exp* and *AtExp* phrase classes, can be defined using this package.

The package does not actually solve the type equation involved, but rather computes the form of the primitive recursion theorem that would be proved if the equation were solved. It then asserts this theorem as an axiom. Russo is presently working on the automatic derivation of this theorem.

Defining the mutually recursive types generates two theorems - the primitive recursion theorem and its simplified form as an *existence theorem*. From the primitive recursion theorem an induction result can be proven using the function

```
prove_mut_rec_induction_thm: thm -> thm
```

Induction theorems can also be proven for *subsets* of the type structure, using

```
prove_subset_induction_thm: string list -> thm -> thm
```

Functions are also available to prove that the type constructors produce distinct types, and that there is a one-one correspondence between constructed types and constructing

$$
\begin{array}{l}
\vdash \forall \mathit{f\_AtExp\_Con}\ \mathit{f\_AtExp\_Paren}\ \mathit{f\_Exp\_AtExp}\ \mathit{f\_Exp\_Con}. \\
\quad \exists_{unique}\ \mathit{fn}. \\
\qquad (\forall C.\ \texttt{OUTL} \circ \mathit{fn} \circ \texttt{INL}\ (\texttt{AtExp\_Con}\ C) = \mathit{f\_AtExp\_Con}\ C) \wedge \\
\qquad (\forall P.\ \texttt{OUTL} \circ \mathit{fn} \circ \texttt{INL}\ (\texttt{AtExp\_Paren}\ P) = \\
\qquad\quad \mathit{f\_AtExp\_Paren}\ (\texttt{OUTR} \circ \mathit{fn} \circ \texttt{INR}\ P)\ P) \wedge \\
\qquad (\forall A.\ \texttt{OUTR} \circ \mathit{fn} \circ \texttt{INR}\ (\texttt{Exp\_AtExp}\ A) = \\
\qquad\quad \mathit{f\_Exp\_AtExp}\ (\texttt{OUTL} \circ \mathit{fn} \circ \texttt{INL}\ A)\ A) \wedge \\
\qquad (\forall C.\ \texttt{OUTR} \circ \mathit{fn} \circ \texttt{INR}\ (\texttt{Exp\_Con}\ C) = \mathit{f\_Exp\_Con}\ C)
\end{array}
$$

Figure 4.5: Primitive Recursion Theorem for the example grammar

$$
\begin{array}{l}
\vdash \forall \mathit{f\_AtExp\_Con}\ \mathit{f\_AtExp\_Paren}\ \mathit{f\_Exp\_AtExp}\ \mathit{f\_Exp\_Con}. \\
\quad \exists\ \mathit{AtExp\_fn}\ \mathit{Exp\_fn}. \\
\qquad (\forall C.\ \mathit{AtExp\_fn}\ (\texttt{AtExp\_Con}\ C) = \mathit{f\_AtExp\_Con}\ C) \wedge \\
\qquad (\forall P.\ \mathit{AtExp\_fn}\ (\texttt{AtExp\_Paren}\ P) = \mathit{f\_AtExp\_Paren}\ (\mathit{Exp\_fn}\ P)\ P) \wedge \\
\qquad (\forall A.\ \mathit{Exp\_fn}\ (\texttt{Exp\_AtExp}\ A) = \mathit{f\_Exp\_AtExp}\ (\mathit{AtExp\_fn}\ A)\ A) \wedge \\
\qquad (\forall C.\ \mathit{Exp\_fn}\ (\texttt{Exp\_Con}\ C) = \mathit{f\_Exp\_Con}\ C)
\end{array}
$$

Figure 4.6: Existence theorem for the example grammar

values. Lastly, it is possible to automatically define recursive functions over the structures, using the existence theorem and the function

```
new_mut_recursive_definition: (string # string # int) list ->
        thm -> string -> term -> thm
```

Figure 4.5 shows the primitive recursion theorem for the types shown in Figure 4.4. The presence of the operators INL, OUTL, INR and OUTR makes the theorem effectively indecipherable, even for such a simple type structure.

The `Expression_INDUCT` theorem is of more interest to us. The subset induction theorem `Exp_INDUCT` illustrates the same problem we had in Section 4.2. It does not allow the necessary assumptions to be made when proving the base cases. However, the total induction theorem is different. It proves that *two* predicates hold simultaneously as we induct over the type structure - one for each type. When proving $P\_Exp(\mathsf{Exp\_AtExp}A)$, we may assume P_AtExpA, and likewise for the other recursions. This assumption could not be made using other solutions to the mutually recursive types problem.

4.3.1 Using the Russo Package in HOL88

Russo's package is implemented in HOL90, the Standard ML version of the HOL system. Although the logics of the two systems are the same, the differences in the meta languages

```
Expression_INDUCT  ⊢ ∀P_AtExp P_Exp.
                       (∀C. P_AtExp (AtExp_Con C)) ∧
                       (∀P. P_Exp P ⊃ P_AtExp (AtExp_Paren P)) ∧
                       (∀A. P_AtExp A ⊃ P_Exp (Exp_AtExp A)) ∧
                       (∀C. P_Exp (Exp_Con C)) ⊃
                       (∀exp. P_AtExp exp) ∧ (∀atexp. P_Exp atexp)

Exp_INDUCT         ⊢ ∀P_Exp.
                       (∀A. P_Exp (Exp_AtExp A)) ∧
                       (∀C. P_Exp (Exp_Con C)) ⊃
                       (∀exp. P_Exp exp)
```

Figure 4.7: Induction theorems for the example grammar

complicate the use of the package for this project.

HOL90 is not as yet completely operational, and thus a decision was made not to move the entire project into that environment. However, because of the amount of code involved in the Russo package ($\sim$ 5000 lines), it was also not feasible to translate the entire package into HOL88. Instead, the syntactic definition are performed in two stages - the first in HOL90 and the second in HOL88. In HOL90 we define the relevant mutually recursive type structure and produce the primitive recursion theorem and the induction theorem for the types. The types and type constructors for the syntax are then introduced into HOL88 using `new_type` and `new_constant`. The primitive recursion and induction theorems are asserted as axioms in HOL88 to give structural semantics to these types. These are sufficient to give us all the properties of the types that are needed.

This approach is justifiable on the grounds that the logics of the two systems are identical. There is no possibility that logical inconsistency will be introduced into the system, given the assumption that the theorems produced by the HOL90 system are consistent.

Although most of the Russo package is concerned with the type definitions themselves, some functions from the package are needed to actually use the the primitive recursion and induction theorems in HOL88. The following functions were ported:

```
prove_subset_existence_thm: string list -> thm -> thm
prove_constructors_one_one: thm -> thm
prove_mut_rec_constructors_distinct: string -> thm -> thm
new_mut_recursive_definition: (string # string # int) list ->
     thm -> string -> term -> thm
```

The last of these is particularly important, as the ability to define recursive functions over the type structures is essential.

During the porting and use of these functions, several bugs were found and fixed in them. This project was the first major use of the package aside from the tests performed by the author himself, and as such was of considerable value in verifying the correctness of the package.

4.4 Defining the Syntax

With the ability to define mutually recursive types in place, it is a straight forward process, if somewhat tiresome, to define the syntactic constructs for Standard ML in HOL. The HOL90 session took several hours to compute the abstract axiom and prove the existence theorem, and consumed a maximum of 60 MB of memory. The resulting axiom pretty-printed to some 300 lines, the existence theorem to 100 and the induction theorem to 75. Fortunately it is rare that we have to work with these theorems directly. Parts of the resulting theory `SML_syntax` are shown in Appendix E.

The theory introduces 16 types and over 50 associated constructors, shown in Appendix A. Instances of these types are created using the constructors. For instance, the constructor Dec_Seq for sequenced declarations has type $Dec \rightarrow Dec \rightarrow Dec$, and a declaration constructed from two sub-declarations would be "`Dec_Seq` *dec1 dec2*".

4.5 Parsing, Pretty-Printing and Manipulating

As with the primitive terms of the HOL logic, it is a tedious exercise to construct instances of the syntax types by hand. For instance, the simple function application `f x` is represented by the HOL term

```
"Exp_Appl (Exp_AtExp (AtExp_Var (Var `f`))) (AtExp_Var (Var `x`))"
```

We obviously need a parser to construct these terms automatically. Hutchins developed a simple parser, and a modified version of it was used for this project.

The size of the terms also makes it difficult to interactively view and manipulate them. A pretty-printer is part of the solution to this problem. Pretty printing *must* be implemented at a very low level of a system such as HOL to enable us to intercept every print of a code fragment. HOL provides a pretty printing library, but it is unfortunately extremely slow. To solve this problem we follow the example of Gordon in [8] and modify the underlying Lisp implementation of the standard HOL pretty printer. The modifications, though lengthy, are routine and do not need to be described here in detail. Lastly, we need a set of functions which will allow us to manipulate program fragments. We use the same naming convention as HOL, thus the destructor function for the constructor Exp_Appl will be `dest_Exp_Appl`.

The following session illustrates the use of all of these features:

4

```
# pretty_on();;
# prs_Exp `f x`;;
"f x" : term

# type_of (prs_Exp `f x`);;
":Exp" : type

# dest_Exp_Appl (prs_Exp `f x`);;
("f", "x") : (term # term)

# pretty_off();;
# prs_Exp `f x`;;
"Exp_Appl(Exp_AtExp(AtExp_Var(Var `f`)))(AtExp_Var(Var `x`))" : term

# dest_Exp_Appl (prs_Exp `f x`);;
("Exp_AtExp(AtExp_Var(Var `f`))", "AtExp_Var(Var `x`)") : (term # term)
```

Chapter 5

Representing The Semantic Types

In the last chapter the syntactic types used in the Formal Definition were successfully embedded into HOL. The next task is to embed the semantic types in a similar fashion. As with the syntactic types, we shall begin our discussion of the semantic types by analyzing some of the problems that were present in Hutchins' original representation of them.

5.1 Finite Partial Functions

One major problem with the original representation of the semantic types has already been identified, for the definitions of these types are mutually recursive in a similar fashion to the syntactic types. This problem shall be discussed further at the end of this chapter. In this section a second problem is identified - the use of lists to represent finite mappings. The solution to this problem is presented in Section 5.2. For the moment we will content ourselves with specifying the properties that a correct representation should have.

5.1.1 Previous Representations

Many of the semantic objects defined in the Formal Definition are "finite maps", $A \xrightarrow{\text{fin}} B$, for some domain A and range B. We shall refer to these as *finite partial functions*, or *fpf* for short. Finite partial functions occur in four different contexts:

$$\begin{aligned}
\mathit{Record} &\equiv \mathit{Lab} \xrightarrow{\text{fin}} \mathit{Val} \\
\mathit{VarEnv} &\equiv \mathit{Var} \xrightarrow{\text{fin}} \mathit{Val} \\
\mathit{ExConEnv} &\equiv \mathit{ExName} \xrightarrow{\text{fin}} \mathit{ExCon} \\
\mathit{Mem} &\equiv \mathit{Addr} \xrightarrow{\text{fin}} \mathit{Val}
\end{aligned}$$

Nearly every evaluation of a Standard ML expression reduces to operations over these objects, and nearly all semantic values make use of them. For example, the paired value (100,200) is represented by the record:

$$\{1_{Lab} \mapsto 100_{Val},\ 2_{Lab} \mapsto 200_{Val}\}$$

In [12], Hutchins correctly recognises that these maps are functions from a finite subset of a (perhaps infinite) domain to a range set. As functions in HOL are "total", he rejects the use of a function of type $A \to B$ to represent finite maps, for then no distinction could be made between elements with mappings and elements without. As a step towards a representation, he defines a new type *fin_result* to represent the possible results of applying an element to a finite map:

$$\mathit{fin\_result}_\alpha = \mathsf{FAILURE} \mid \mathsf{RESULT}\ \alpha$$

Here FAILURE represents an undefined result, and RESULT is a constructor for the result of a successful application. The type $\mathit{fin\_result}_B$ is then the set of possible results for any finite map with range B.

A partial function can then be represented as a function from the domain set to *fin_result*:

$$A \longrightarrow \mathit{fin\_result}_B \tag{5.1}$$

However, this representation was rejected by Hutchins on the grounds that it "makes it difficult to write down a finite map explicitly". As a solution to this problem, Hutchins decided to represent the maps using lists of pairs. Thus the map $\{1 \mapsto 6\}$ would be represented by $\mathsf{fin}[(1,6)]$. To quote:

> This representation is easiest for the user because lists are well supported in HOL, with a variety of predefined functions and pre-proved theorems.

This solution initially seems to work quite well. The definitions of application and extension come easily, the semantics being that if a particular element x in the domain has more than one pair (x, y) in the list, then the first such y is the result of applying x to the map. This representation was sufficient for Hutchins for the remainder of his work.

Unfortunately, this solution to the representation problem is quite badly flawed. Like all functions, finite partial functions are *unstructured* - there is no ordering of the elements in the domain. Furthermore, the mapping between lists and functions is not isomorphic - in general each function has several possible representations as a list. It is a simple fact that finite partial functions are *not* lists of pairs, no matter which way we look at it. Lists aren't a "logically correct" representation of a finite map any more than a set is a logical representation for a list, or a bag is a logical representation for a set. This can be illustrated by considering the following equality between two finite partial functions represented as lists:

$$\mathsf{fin}\ [(1,100);\ (1,200)] = \mathsf{fin}\ [(1,100)]$$

Given that the fin constructor is 1-1, it would be a simple task in HOL to prove that this equality does not hold, for the underlying lists are different. However, semantically the finite partial functions represented by the two lists are the same, as they both contain only one mapping $\{1 \mapsto 100\}$.

The experience of trying to state and prove an induction theorem using the initial representation quickly led to the recognition that a "correct" representation was needed. Foremost, it is crucial that finite maps which are semantically the same should be provably equivalent in HOL.

It was surprisingly difficult to detect that lists actually make for a bad representation, for their use does seem to have some merit. The cause of this problem would seem to be rooted in our experiences as computer programmers. In general, we are not used to having the logical tools available to precisely specify the properties of the types we want. This is especially true with "unstructured" types such as sets and functions. Using lists is an obvious implementation step, which appears correct to the programmer but is logically incorrect. HOL both demands and allows logical accuracy in representing such constructs.

5.1.2 Requirements of a Theory of Finite Partial Functions

It is worth considering if the representation illustrated in Equation 5.1 is a suitable solution to our problem. The problem here is again subtle. If we defined a new type:

$$\mathit{fin}_{\alpha,\beta} = \alpha \rightarrow \mathit{fin\_result}_{\beta}$$

then we must ask if an arbitrary map f of this type has the properties we require. When the question is posed as such, we can see immediately that the answer is no, for there is no requirement that any object of the new type have finite domain. Although any particular instance we create in the semantics may happen to be finite, the type now contains partial functions in general, including ones whose domains are infinite. It would be possible to use this representation, but each time we wanted to induct over the domain of the function the additional requirement that the domain were finite would need to be proved.[1]

The properties that are required of the type may be summarized as follows:

- The set of finite maps $\alpha \xrightarrow{\text{fin}} \beta$ is the smallest set of functions from α to $\mathit{fin\_result}_{\beta}$ containing both the empty map and which is closed under extension by single maplets $\{a \mapsto b\}$. We will use ZIP for the empty map and EXTfor map extension.
- Applying a finite map f to a value a results in either a special value FAILURE or the value that a is mapped to in f.
- Two finite maps f and f' are equal if and only if for every $a \in \alpha$ the result of applying a to f is the same as the result of applying a to f'.

Defining these well known properties of partial functions may seem to be stating the obvious, at least to the mathematician. However, the way we represent these objects in HOL *must* be precise. In other words, the finite maps must have these properties exactly. We may thus use this set of properties as a litmus test for theories of the maps.

5.2 A HOL Theory of Finite Partial Functions

The task in the previous section was simple - the incorrect representation of finite partial functions was identified and then the properties of a solution were specified. In this section we describe the construction of a HOL theory of the new type.

5.2.1 Defining the Representation

As finite partial functions are not concrete types, the only useful type construction method available to us is `new_type_definition`, described in Section 4.1. Our first task is to find a suitable representation set for our finite maps from among existing types. A new type, isomorphic to this set, is then brought into existence by the specification rule of the system.

[1] A similar problem occurs with the *set* library in HOL, hence HOL also provides a second library defining *finite sets*.

After this, our first aim is to "lift" the properties of the representation set to the level of the constructed type via the isomorphic mappings. We have already touched on several possible representations for maps from α to β:

1. Functions from α to β + FAILURE, with a restriction to ensure that only a finite number of elements are mapped into β.

2. Finite sets of pairs (x, y) where $x \in \alpha$ and $y \in \beta$. For each $x \in \alpha$ a set could contain at most one pair (x, y).

3. Some representation using lists where we are ensured that there is exactly one list corresponding to each possible finite partial function

. The first representation was chosen, although the second would also work adequately. We begin by making definitions for the representations of the empty map ZIP and the map extension operator EXT:[2]

1

```
# new_definition "ZIP_REP x = FAILURE";;
ZIP_REP_DEF = |- ZIP_REP z = FAILURE

# new_definition "EXT_REP (x,y) f z = (x = z) => RESULT y | f z";;
EXT_REP_DEF = |- EXT_REP (x,y) f z = (x = z) => RESULT y | f z
```

The constants FAILURE and RESULT have been mentioned before with respect to the type *fin_result*. ZIP_REP has type *REP* where *REP* is $\alpha \to fin\_result_\beta$. EXT_REP has type $(\alpha \times \alpha) \to REP \to REP$.

We now define the representation set for our type. The set is the smallest set of representation functions which contains both the empty map ZIP_REP and which is closed under extension by EXT_REP. This is stated by the following definition of the predicate IS_fpf_REP.[3]

2

$$\begin{array}{l} \vdash_{def} \ \mathtt{IS\_fpf\_REP}\ f = \\ \quad \forall P.\ (P\ \mathtt{ZIP\_REP}) \wedge \\ \qquad (\forall f'\ x\ y.\ P\ f' \wedge P\ (\mathtt{EXT\_REP}\ (x,y)\ f')) \supset\ P\ f \end{array}$$

We must now prove that the type representation is well founded by showing that the representative set has at least one element. The proof is trivial as we already have one such element, the empty map ZIP_REP. The following theorem is thus easily proven:

3

$$\vdash \exists f.\ \mathtt{IS\_fpf\_REP}\ f \qquad\qquad (\mathtt{fpf\_REP\_EXISTS})$$

The type definition is now introduced by the following calls to `new_type_definition` and `define_new_type_bijections`:

[2] In general we shall not show the HOL calls, but only the theorems they produce. It must be remembered that behind each theorem there is either a definition, or a proof constructed using tactics, rules and goals. For definitions we use the symbol $\vdash_{def}$, and for theorems $\vdash$.

[3] An equivalent definition would have been to state that the cardinality of the set of all elements which did not map to FAILURE was finite.

4

```
# new_type_definition(`fpf`, fpf_REP_EXISTS);;
# define_new_type_bijections `fpf_ISO_DEF` `ABS_fpf` `REP_fpf` fpf_TY_DEF;;
```

These calls define the new type using the primitive type specification rule of the HOL logic. The predicate IS_fpf_REP defines the representing set and the theorem `fpf_REP_EXISTS` provides the support for the definition. The name of the new type is *fpf*. Bijections are set up between the representation set and the new type. ABS_fpf maps from the representation set to the new type, and REP_fpf vice-versa. The one-to-one and onto properties of these mappings are also automatically proven for us. We now "lift" the definitions of the representations of ZIP and EXT to define ZIP and EXT themselves:

5

$$\vdash_{def} \texttt{ZIP} = \texttt{ABS\_fpf ZIP\_REP}$$
$$\vdash_{def} \texttt{EXT}\ (x, y)\ \mathit{fpf} = \texttt{ABS\_fpf}\ (\texttt{EXT\_REP}\ (x, y)\ (\texttt{REP\_fpf}\ \mathit{fpf}))$$

A variety of lemmas associated with these definitions must now be proven. These include the important fact that EXT maintains membership in the representation set. Once these are derived we next prove the central property that is required - a primitive induction principle over the new type. The proof parallels the derivation of the induction principle for the natural numbers given in the HOL system, and we shall not include the details here. The theorem proved is:

6

$$\vdash P\ \texttt{ZIP} \wedge (\forall \mathit{fpf}\ x\ y.\ P\ \mathit{fpf} \wedge P\ (\texttt{EXT}\ (x, y)\ \mathit{fpf})) \supset (\forall \mathit{fpf}.\ P\ \mathit{fpf})$$

Note the weakness of this induction principle - it may not, for instance, be assumed in the step case that the element that is being added is not already in the domain of the finite map. However the theorem is sufficient to ensure that the functions do in fact have finite domain.

The definition of the all important APPLY operator, which is used for function application, can now be made:

7

$$\vdash_{def} \texttt{APPLY}\ x\ \mathit{fpf} = (\texttt{REP\_fpf}\ \mathit{fpf})\ x$$

The definition is somewhat trivial due to our choice of representation - it simply refers back to the representation function and applies the element. The following theorems are now be proven:

8

$$\vdash \texttt{APPLY}\ z\ \texttt{ZIP} = \texttt{FAILURE}$$
$$\vdash \texttt{APPLY}\ z\ (\texttt{EXT}\ (x, v)\ f) = ((x = z) \rightarrow \texttt{RESULT}\ v \mid \texttt{APPLY}\ x\ f)$$

These two theorems were part of our specification of the properties necessary for finite maps. They are also proven in a variety of more useful forms. The last central theorem that we need to prove concerns equality:

9

$\vdash$ ($f1$ = $f2$) = ($\forall x$. APPLY x $f1$ = APPLY x $f2$)

This states that two *fpf*s are equal if and only if they are equal for every application, and is proved by reference to the representation. Some interesting and useful theorems that can now be proven are:

10

$\vdash$ $\neg$(EXT (x,y) f = ZIP)
$\vdash$ EXT (x,u) (EXT (x,v) f) = EXT (x,u) f
$\vdash$ $\neg$(x = y) $\supset$ (EXT (x,u) (EXT (y,v) f) = EXT (y,v) (EXT (x,u) f))
$\vdash$ (EXT (x,y) ZIP = EXT (x',y') ZIP) $\supset$ (x = x') $\wedge$ (y = y')
$\vdash$ SUCCEEDS (APPLY x fpf) = ($\exists_{unique}$ y. EXT (x,y) fpf = fpf)

The last theorem is quite strong - it states that each element for which APPLY succeeds has one and only one extension which does not disturb the map.

5.2.2 More Definitions

The initial aim of this section has been achieved - a new type with the required properties. A handful of useful theorems about the type are also available. However, the theory is still in a primitive state, for it has not been enriched in any way with further helpful definitions.

EXTBY

The first definition we make is the *sum* of two finite maps. Conflicts are resolved by choosing the mappings in the first in preference to those in the second. We call the addition operator EXTBY, though perhaps the infix ++ would have been better. To make the definition, we could resort back to the underlying representation. However, this is bad technique, for a complete set of primitives for the type have just been constructed. Using the primitives makes the theory largely independent of the underlying representation.

As finite partial functions are an unstructured type, there is no recursive function specification technique available to use. HOL offers two main choices - to use the ε-choice operator, or the primitive constant specification rule. We choose the former:

11

EXTBY_P = $\vdash_{def}$ EXTBY_P sum map map' =
($\forall x$. APPLY x sum = (SUCCEEDS(APPLY x map) $\rightarrow$ APPLY x map | APPLY x map'))
EXTBY_DEF = $\vdash_{def}$ EXTBY map map' = (εsum. EXTBY_P sum map map')

The first definition specifies the predicate EXTBY_P, which will be true only when *sum* is the addition of *map* and map'. EXTBY is then defined using the ε operator. A value for *sum* is chosen which satisfies the predicate EXTBY_P, unless there is no such value in which case an arbitrary value is chosen.

Using the ε-choice operator in the fashion above is a common construct in HOL for complicated definitions. However, it is virtually impossible to manipulate the operator in general proofs, so care must be taken to produce a theorem which eliminates it as soon as possible. In the above example, the EXTBY_P predicate is in fact one-one, in the sense that, given *map* and *map'*, there will be exactly one *sum* for which EXTBY_P is true. Thus the ε operator will always choose this value. The fact that at least one value satisfies the predicate is stated in the following way:

12

$\vdash\ \ \forall map\ map'.\ \exists sum.\ \texttt{EXTBY\_P}\ sum\ map\ map'$

This theorem is proved by induction down the first map. It provides sufficient justification for the following theorem which eliminates the ε operator:

13

$\vdash\ \ \forall map\ map'.\ \texttt{EXTBY\_P}\ (\texttt{EXTBY}\ map\ map')\ map\ map'$

That is to say, EXTBY_P is true for the value chosen by EXTBY. By rewriting with the definition of EXTBY_P we now have:

14

$\vdash \texttt{APPLY}\ x\ (\texttt{EXTBY}\ map\ map') =$
$\quad(\texttt{SUCCEEDS}(\texttt{APPLY}\ x\ map) \rightarrow\ \texttt{APPLY}\ x\ map \mid \texttt{APPLY}\ x\ map')$

Combining this result with the equality theorem gives the final theorems for EXTBY:

15

$\texttt{EXTBY\_ZIP} = \vdash \texttt{EXTBY ZIP}\ f = f$
$\texttt{EXTBY\_EXT} = \vdash \texttt{EXTBY (EXT}\ (x,y)\ f1)\ f2 = \texttt{EXT}\ (x,y)\ (\texttt{EXTBY}\ f1\ f2)$

The details of the tactics and proof techniques used in deriving these theorems have been omitted for the sake of brevity. However, the style of derivation seen above is used time and time again in HOL to introduce complex definitions such as EXTBY. Some argue that the ε operator should not be used unless we are directly invoking the mathematical Axiom Of Choice, and that the primitive specification rules should be preferred instead. However, the use of the ε operator is in fact easier and clearer in most cases.

TRANSFORM and DOM

This section describes two more important operations over finite maps - calculating the domain and transforming the range. The full details of the derivations are not shown, but only the definitions and the principal theorems proven.

The primitive definitions for the DOM operator are:

16

$\vdash_{def} \texttt{DOM\_P}\ dom\ map = (\forall x.\ x\ \texttt{IN}\ dom = \texttt{SUCCEEDS(APPLY}\ x\ map))$
$\vdash_{def} \texttt{DOM}\ map = (\varepsilon dom.\ \texttt{DOM\_P}\ dom\ map)$

The basic derived results:

17

$\vdash$ `DOM_P (DOM` *map*`)` *map*
$\vdash$ `DOM ZIP = EMPTY`
$\vdash$ `DOM (EXT (`x,y`)` *fpf*`) =` x `INSERT (DOM` *fpf*`)`

Some useful theorems:

18

$\vdash$ `(DOM` f `= EMPTY) = (`f `= ZIP)`
$\vdash$ x `IN (DOM` *fpf*`) = (`$\exists y$`. APPLY` x *fpf* `= RESULT` y`)`
$\vdash$ $\neg x$ `IN (DOM` *fpf*`) = (APPLY` x *fpf* `= FAILURE)`

Some important theorems about the cardinality of the domain:[4]

19

$\vdash$ $\forall f$`. FINITE(DOM` f`)`
$\vdash$ `(CARD(DOM` f`) = SUC` i`)` $\supset$
  `(`$\exists f'$ x y`.` $\neg$`SUCCEEDS(APPLY` x f'`)` $\wedge$ `(CARD(DOM` f'`) =` i`)` $\wedge$ `(`f `= EXT(`x,y`)`f'`))`

The function TRANSFORM has as its first argument a (total) function which maps elements of the range of a map to new values. It is defined in the usual fashion using the ε-choice operator:

20

$\vdash_{def}$ `TRANSFORM_P` *map'* *fn* *map* =
  `(`$\forall x$`. APPLY` x *map'* =
    `(SUCCEEDS(APPLY` x *map*`)` $\rightarrow$ `RESULT(`*fn*`(RESULTOF(APPLY` x *map*`)))`
                                  `| FAILURE))`
$\vdash_{def}$ `TRANSFORM` *fn* *map* = `(`ε*map'*`. TRANSFORM_P` *map'* *fn* *map*`)`

The basic derived results are:

21

$\vdash$ `TRANSFORM_P (TRANSFORM` *fn* *map*`)` *fn* *map*
$\vdash$ `TRANSFORM` *fn* `ZIP = ZIP`
$\vdash$ `TRANSFORM` *fn* `(EXT (`x,y`)` *fpf*`) = EXT (`x,*fn* y`) (TRANSFORM` *fn* *fpf*`)`

5.2.3 Stronger Induction Principles

We conclude the discussion of the construction of this theory by describing the derivation of two stronger induction principles for finite maps.

A commonly used induction technique on finite maps is to induct over the cardinality of the domain of the maps. In the step case it is useful to be able to assume the predicate that is being proven for all maps with smaller domain size. This result is based on the strong induction principle for the natural numbers:

[4] CARD and FINITE are both defined in the HOL *set* library.

22

$\vdash P\ \mathtt{0} \wedge (\forall j.\ (\forall i.\ i \leq j \supset\ P\ i) \supset\ P(\mathtt{SUC}\ j)) \supset\ (\forall n.\ P\ n)$

When instantiated to the particular case of induction over the size of a domain this theorem becomes:

23

$$
\begin{array}{l}
\vdash\ (P\ \mathtt{ZIP}) \wedge \\
\quad (\forall j. \\
\qquad (\forall i\ \mathit{fpf}.\ i \leq j \wedge \mathtt{CARD(DOM}\ \mathit{fpf}) \leq i \supset\ P\ \mathit{fpf}) \supset \\
\qquad (\forall \mathit{fpf}.\ \mathtt{CARD(DOM}\ \mathit{fpf}) \leq (\mathtt{SUC}\ j) \supset\ P\ \mathit{fpf})) \supset \\
\quad (\forall \mathit{fpf}.\ P\ \mathit{fpf})
\end{array}
$$

The step case states that, given a natural number j and the assumption that P is true for all cardinalities less than or equal to j, we must prove that P is true for all maps with cardinality $\leq j+1$.

The next induction principle is proved by using the principle just derived. It is similar to the very first induction principle, except it allows us to make the additional assumption that the element being used to extend the map in the step case is not in the domain of the map:

24

$$
\begin{array}{l}
\vdash\ (P\ \mathtt{ZIP}) \wedge \\
\quad (\forall \mathit{fpf}\ x\ y.\ P\ \mathit{fpf} \wedge \neg\mathtt{SUCCEEDS(APPLY}\ x\ \mathit{fpf}) \supset\ P(\mathtt{EXT}(x,y)\mathit{fpf})) \\
\supset\ (\forall \mathit{fpf}.\ P\ \mathit{fpf})
\end{array}
$$

The induction principles are needed to complete the derivation of the fundamental theorems for the following operators:

RANGE *map*	The set of all values mapped to by some domain element
UNEXT x *map*	Removes the element x from the domain of *map*
EVERYF P *map*	True if P is true for every mapping in *map*

We shall not give the details of these derivations, though they are by no means trivial. The UNEXT operator is particularly powerful as it allows us to decompose maps.

In all, over 60 theorems are proved and stored with the *fpf* theory. Lemmas not useful outside the construction of the theory are not saved. Approximately 1500 lines of HOL code are needed to construct the theory.

5.3 The Semantic Types and More Mutual Recursion

5.3.1 Mutual Recursion through Finite Partial Functions

Even with a well constructed theory of finite partial functions, a problem still remains with constructing the semantic type structures that are needed for the Formal Definition. The problem, illustrated in Figure 5.1, is a new manifestation of the mutually recursive types problem, induced by the use of finite partial functions in our theories.

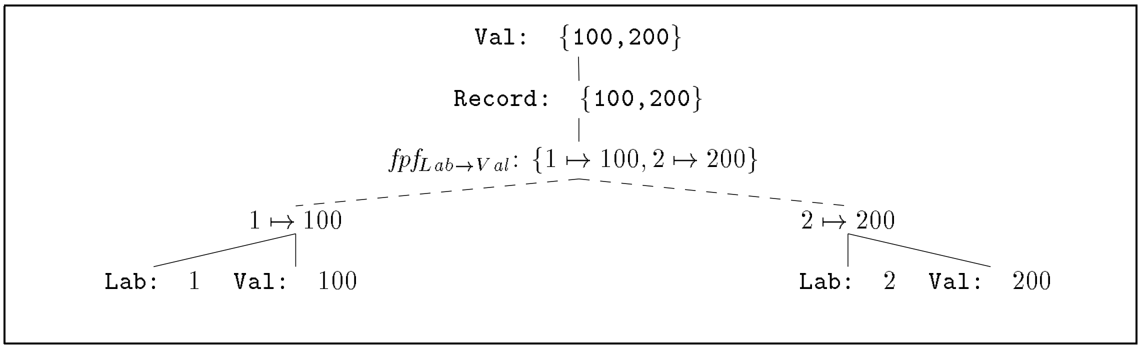

Figure 5.1: A Value displaying mutual recursion through a finite partial function

The Formal Definition's specification of the *Val* type, of which Figure 5.1 is an instance, is shown in figure 5.2. Note that the type *Val* is defined in terms of the type *Record*, which is defined as a finite partial function from type *Lab* to type *Val*.

The mutual recursion package developed by Russo is not capable of solving the type equation presented here. The reason is that only concrete mutual recursion is solved, and not mutual recursion through an unstructured type such as a finite map. To demonstrate this problem further, consider the following type definitions:

$$
\begin{array}{rcll}
data & = & \mathsf{INT}\ num & \\
 & | & \mathsf{RECORD}\ num\ data & (5.2) \\
data & = & \mathsf{INT}\ num & \\
 & | & \mathsf{RECORD}\ (num \times data)\, list & (5.3) \\
data & = & \mathsf{INT}\ num & \\
 & | & \mathsf{RECORD}\ (num \xrightarrow{\text{fin}} data) & (5.4)
\end{array}
$$

The structured type definition packages succeed for specification 5.2 as it is a simple recursion on the type being defined. However, there are no existing packages which can solve either 5.3 or 5.4 automatically. The packages have sufficient structure to solve 5.3 with some minor modifications, for this type has concrete structure. No existing packages have sufficient internal structure to solve the unstructured type given by 5.4.

Constructing a complete solution to the problem of recursive unstructured types is complex, and well beyond the scope of this thesis. In principal, it should be possible to develop techniques which allow recursive types including functions to be defined. As our finite partial functions are really just a restricted class of functions, this would solve our problem. However, as the main aim of this project is to develop a theory of Standard ML, we will be content with asserting the semantics of the types as axioms, once we have investigated methods of representing the recursion.

$$
\begin{array}{rcl}
\ldots & = & \ldots \\
\mathit{Val} & = & \{\texttt{:=}\} \cup \mathit{SVal} \cup \mathit{BasVal} \cup \mathit{Con} \\
& & \cup (\mathit{Con} \times \mathit{Val}) \cup \mathit{ExVal} \cup \mathit{Record} \cup \mathit{Addr} \cup \mathit{Closure} \\
\mathit{Record} & = & \mathit{Lab} \xrightarrow{\text{fin}} \mathit{Val} \\
\ldots & = & \ldots
\end{array}
$$

Figure 5.2: Definitions of the types *Val* and *Record*

5.3.2 Finding a Solution

As a first step to a solution, it is instructive to consider an example of what the principal theorems look like for the type

$$
\begin{array}{rcl}
\mathit{data} & = & \mathsf{INT}\ \mathit{num} \\
& | & \mathsf{RECORD}\ (\mathit{num} \times \mathit{data})\mathsf{list}
\end{array}
$$

The HOL details of the derivation of this type are not shown here, though the derivation was performed to investigate the problem. The fundamental theorem for the type is:

25

$$
\begin{array}{l}
\vdash \forall f1\ f2. \\
\quad \exists_{unique} fn. \\
\quad\quad (\forall n.\ fn\ (\texttt{INT}\ n) = f1\ n) \wedge \\
\quad\quad (\forall list.\ fn\ (\texttt{RECORD}\ list) = fn\ (\texttt{MAP}\ (fn \circ \texttt{SND})\ list)\ list)
\end{array}
$$

The function is a normal primitive recursion theorem asserting the existence of a unique recursive function over the type, given a specification. The induction theorem for the type is:

26

$$
\begin{array}{l}
\vdash (\forall n.\ P\ (\texttt{INT}\ n)) \wedge \\
\quad (\forall list.\ \texttt{EVERY}\ (\texttt{MAP}\ (P \circ \texttt{SND})\ list) \supset\ P\ (\texttt{RECORD}\ list)) \supset \\
\quad (\forall data.\ \texttt{P}\ data)
\end{array}
$$

The functions MAP and EVERY come from the HOL list theory and have the usual semantics. The form of the second conjunct (the step case) is interesting. It allows us to assume the predicate P for instances of the type *data* in the list *list* when proving $P(\mathsf{RECORD}\ \mathit{list})$.

Using these as our guide, and without further justification, it would seem reasonable that the primitive recursion and induction axioms for the type

$$
\begin{array}{rcl}
\mathit{data} & = & \mathsf{INT}\ \mathit{num} \\
& | & \mathsf{RECORD}\ (\mathit{num} \xrightarrow{\text{fin}} \mathit{data})
\end{array}
$$

should be as follows:

27

$$\vdash \forall f1\ f2.\ \exists_{unique} fn.\ (\forall n.\ fn\ (\mathtt{INT}\ n) = f1\ n) \wedge (\forall map.\ fn\ (\mathtt{RECORD}\ map) = f2\ (\mathtt{TRANSFORM}\ fn\ map)\ map)$$

$$\vdash (\forall n.\ P\ (\mathtt{INT}\ n)) \wedge (\forall fpf.\ \mathtt{EVERYF}\ (P \circ \mathtt{SND})\ fpf \supset P\ (\mathtt{RECORD}\ fpf)) \supset (\forall data.\ P\ data)$$

The functions TRANSFORM and EVERYF were described in Section 5.2, and replace MAP and EVERY in the definitions. The induction axiom shown above is certainly correct, in the sense that it should be derivable from a proper derivation of the type. As before it allows us to assume that P is true for all instances of the type *data* occurring within the finite map *fpf* when proving the step case. The primitive recursion axiom is more questionable, as it allows us to specify recursive functions which arbitrarily transform the finite maps occurring in the recursive types. However, this seems reasonable, for transformation of the range does not add extra structural properties to a finite partial function.[5] The primitive recursion property is used only once in this project, when we derive a general primitive recursion theorem for a Standard ML type in Chapter 9.

5.3.3 Defining the Semantic Types (at last)

In the last section we developed a way to characterise a mutually recursive type structure which includes finite partial functions. This is done by asserting the appropriate primitive recursion and induction theorems as new axioms in HOL. Using this technique the semantic types that are needed by the formal definition can now be defined.

The process of defining the semantic types may be summarized as follows:

1. A proper theory of finite partial functions was developed.

2. A version of the semantic types was developed using the Russo mutually recursive types package, using lists to represent the finite partial functions.

3. The theorems produced by the Russo package were modified to replace the list representations with instances of the *fpf* type constructed in step 1.

4. The types, constructors and theorems were then introduced into HOL88 by using `new_axiom` and `new_type`.

The semantic type theory defines 12 types and over 40 associated constructors. The most important semantic type *Val* has constructors Val_Addr, Val_Closure, Val_Con and so on. For instance the value "`Val_Closure (Closure` *match env varenv*`)`" is an instance of the type *Closure*.

[5]Note that TRANSFORM does not effect the domain of a finite partial function, only its range. If transformation could produced a new map by operating on individual mappings, then the primitive recursion axiom outlined above would not be valid.

As with the syntactic types, we construct a pretty printer for these constructs, and also for finite partial functions. We allow the printing of closures to be toggled, for it is usually only when a closure is being used that it needs to be displayed. There is no particular need for a parser for semantic values, for it is rare that they need to be explicitly entered into HOL. Lastly, it is useful tp define a set of syntax operators to construct and destruct the terms made up of these types. The session below shows an example of how these functions are used:

28

```
# let tm = "Val_Closure (Closure m e v)";;
tm = "[m,e,v]" : term      %< pretty-printed version of a closure >%

# is_Val_Closure tm;;
true : bool

# let tm2 = dest_Val_Closure tm;;
tm2 = "Closure m e v" : term

# let tm3 = dest_Closure tm2;;
tm3 = ("m", "e", "v") : (term # term # term)
```

Chapter 6

Representing the Core Semantics

This chapter describes the third and last stage in the process of representing the Formal Definition of Standard ML in HOL. The syntactic and semantic types have been correctly represented, and we now embed the semantics themselves. The Core semantics come in two parts, both of which were described in Chapter 2:

1. The 57 inference rules which together determine the inferability of sentences.

2. The definitions of the base values and initial environments of the language.

The relationship between the theories constructed in this and previous chapters is shown in Figure 6.1. The main product of this chapter is the theory `SML_dynamics`. In it is stored all the definitions which capture the full logical content of the semantics of the Core.

6.1 Problems with Representing the Inference Rules

As with the Standard ML type representations, a preliminary embedding of the dynamic semantics into HOL was performed by Matthew Hutchins. A major part of his work was devoted to describing how various interpretations of the inference rules led to incompleteness in the resulting HOL theory. Hutchins concluded with a representation that he thought to be complete, though it shall shortly be shown that this is not so.

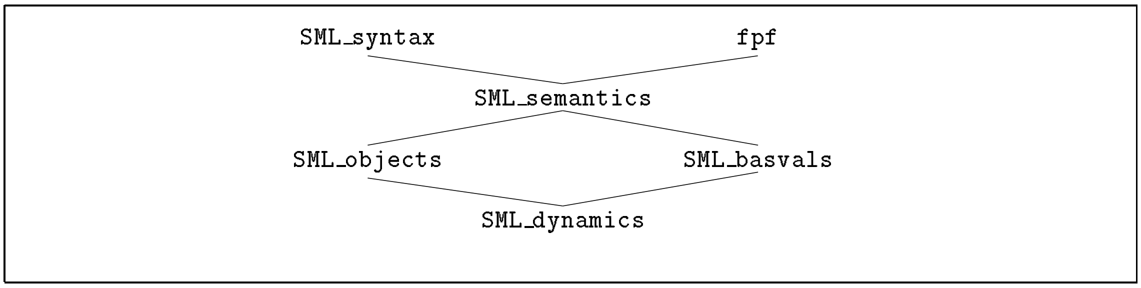

Figure 6.1: The ancestries of the semantic representation theories

6.1.1 Some more definitions

To discuss different HOL representations of the semantics, several more HOL constructs are needed. HOL allows shorthand notations for existing monomorphic types to be defined, and thus it is possible to define:

$$Sentence \equiv State \times Env \times Obj \times Phrase \times Result \times State$$

The sentences of the dynamic semantics are written using the predicate IS, which has type $Sentence \rightarrow bool$. The sentence $s, E \vdash exp \Rightarrow v, s'$ thus corresponds to the HOL term:

```
"IS(s, E, Obj_NONE, Ph_Exp exp, Res_Val v, s')" : term
```

The first and last arguments have type *State*, and the second type *Env*. To allow for other backgrounds, phrases and results, the remaining arguments have type *Obj*, *Phrase* and *Result* respectively. These types are defined by:

$$\begin{array}{rcl} Obj & = & \mathsf{Obj\_NONE} \cup Val \cup Record \\ Phrase & = & AtExp \cup Exp \cup \cdots \cup Pat \\ Result & = & Val \cup Pack \cup Record \cup \{\mathsf{FAIL}\} \cup VarEnv \cup ExConEnv. \end{array}$$

We use the `define_type` mechanism to introduce these into HOL. For example:

```
# define_type `Obj_DEF` `Obj = Obj_NONE
                             | Obj_Val  Val
                             | Obj_Rec  Record `;;
```
1

The full syntax for these types is shown in Appendix A. In this thesis the *Obj*, *Result* and *Phrase* constructors will often be omitted when writing inferable sentences and other constructs, unless ambiguity results. Exception-packets will generally be denoted by `[`*exception-name*`]`.

6.1.2 Incorrect Representations

Implicative

The representation first used by Hutchins was a direct axiomatic modeling of the implicative nature of the Inference Rules. A series of axioms is asserted into in HOL, each being an inference rule represented as an implication from the premises to the conclusion. For example, consider Rule 135, which deals with the evaluation of value bindings in declarations:

$$\frac{E \vdash exp \Rightarrow v \quad E, v \vdash pat \Rightarrow VE}{E \vdash pat \texttt{=} exp \Rightarrow VE}$$

This rule would be asserted as the axiom shown in Figure 6.2.

This theorem must obviously be true for any representation that is built for the semantics. However, it must be questioned if the set of all such axioms constitutes a complete basis for the description of the Core. As Hutchins demonstrated, they do not. The reason for this is that, although not explicitly stated in the definition, a sentence is regarded as inferable if *and only if* it is deducible from the given inference rules. That is to say:

```
⊢  IS(s, E, Obj_NONE, Ph_Exp exp, Res_Val val, s') ∧
    IS(s', E, Obj_Val val, Ph_Pat pat, Res_VarEnv VE, s'') ⊃
    IS(s, E, Obj_NONE, Ph_ValBind(ValBind pat exp), Res_VarEnv VE, s'')
```

Figure 6.2: The Implicative Form of the ValBind Axiom

- Any sentence which follows from the inference rules is a valid sentence.
- Any other sentence is not.

A statement which relies on the second condition is:

> Given the sentence $E \vdash (atexp) \Rightarrow v$, we may deduce $E \vdash atexp \Rightarrow v$ for some value v.

Though obviously true, the statement cannot be proven from the implicative axioms, for it relies on the fact that there happen to be no other ways to generate a sentence of the form on the left. This is a meta-statement about the logic, and is not expressed in the axioms of the implicative representation. In addition, there is no obvious easy way of asserting extra axioms to tighten up this definition.

Equational

In response to this problem, it may seem logical to replace the implications in the axioms with assertions of equality between the premises and conclusion. However, in certain cases, there are several rules which produce sentences of the same form. Most commonly, whenever an inference rule may be effected by exceptions, several cases arise where exceptions in the sub-phrases cause the parent phrase to evaluate to an exception. Hutchins' ultimate solution to this problem is to characterise, in one axiom per phrase category, all the possible inferable results for that category. For instance, the implicative axiom in Figure 6.2 is combined, along with several others, into the axiom shown in Figure 6.3.

The first disjunct on the right of the equality corresponds to theorem in Figure 6.2. The use of disjuncts on the right makes the axiom "closed", for to produce the inferable sentence shown on the left, one of the disjuncts must have been true.

This representation seems complete. Each axiom is closed, explicitly listing the sub-conditions that must be true for the sentences of a particular type to be inferable. To quote Hutchins' appraisal:

> It is fairly easy to see that these axioms are complete. For each possible evaluation there is precisely one disjunct belonging to one of the axioms giving the conditions under which the evaluation can be made. Impossible evaluations are ruled out because the inferable sentence defined by each axiom is in its most general form and related to its evaluation by equality.

$$
\begin{array}{l}
\vdash \ \forall s\ E\ pat\ exp\ res\ s'. \\
\quad \texttt{IS}(s,\ E,\ \texttt{Obj\_NONE},\ \texttt{Ph\_ValBind (ValBind } pat\ exp\texttt{)},\ res,\ s') = \\
\quad (\exists VE\ val\ s''. \\
\qquad (res = \texttt{Res\_VarEnv } VE) \wedge \\
\qquad \texttt{IS}(s,E,\texttt{Obj\_NONE},\texttt{Ph\_Exp } exp,\texttt{Res\_Val } val,s'') \wedge \\
\qquad \texttt{IS}(s'',E,\texttt{Obj\_Val } val,\texttt{Ph\_Pat } pat,\ \texttt{Res\_VarEnv } VE,\ s')) \vee \\
\quad (\exists val\ s''. \\
\qquad (res = \texttt{Res\_Pack [`Bind`]}) \wedge \\
\qquad \texttt{IS}(s,E,\texttt{Obj\_NONE},\texttt{Ph\_Exp } exp,\texttt{Res\_Val } val,s'') \wedge \\
\qquad \texttt{IS}(s'',E,\texttt{Obj\_Val } val,\texttt{Ph\_Pat } pat,\ \texttt{Res\_FAIL},\ s')) \vee \\
\quad (\exists p. \\
\qquad (res = \texttt{Res\_Pack } p) \wedge \\
\qquad \texttt{IS}(s,E,\texttt{Obj\_NONE},\texttt{Ph\_Exp } exp,\texttt{Res\_Pack } p,s'))
\end{array}
$$

Figure 6.3: The Equational Form of the ValBind Rule

Why the Equational Representation is Incomplete

It is unfortunate that the axioms are, in fact, not complete, and that the line of reasoning given above is not sufficient to guarantee the results we need. The problem lies with infinite computations. Consider applying the following degenerate Standard ML function to the value 1:

```
val rec f = fn x => f x
```

The function will obviously compute indefinitely due to the recursive call on `f`. As such, it should be possible to show that the sentence

$$s, obj \vdash P \Rightarrow res, s' \tag{6.1}$$

is not inferable for any s, obj, res and s'. Using Hutchins' axioms, only one possible path of proof is possible - to start with the axiom which deals with the evaluation of function applications. The proof eventually discovers that the pattern match for the function succeeds (in the sense that a result could be inferred for it) and that the evaluation of variables `f` and `x` also succeed. Thus the only way that no result could be inferred for the whole computation was if no result was possible for applying 1 to `f`. The proof has become degenerate, for it has been reduced to the case of what is trying to be proved in the first place. The only way that it can be shown that no result is inferable for `f x` is by assuming that very statement.

The essential problem is that using the equational our axioms *do not capture the restriction that the inference process must be finite*. It is true that each axiom holds as an equality, for if one disjunct of the right hand side holds, then by a rule of inference the conclusion must hold. Likewise, if the conclusion holds then one of the disjuncts must hold because the inference rules represent all the possible computations. Despite this fact, the equational representation is not sufficient to let all cases of non-inferability be deduced.

The problem occurs not only with degenerate functions such as that used above, but in fact with all infinite computations. The axioms do not allow us to conclude anything about computations which do not terminate. For instance, it would be logically consistent to assert that the above computation evaluates to, say, 163, for we have no way of proving that it doesn't.

It may be questioned if it is strictly necessary that the representation of the semantics allow the deduction of non-inferability for non-terminating evaluations. For Hutchins it was not, for his work ended with the symbolic evaluator described in Chapter 8. Evaluation requires only forward rules, and not backward reasoning. However, the problem is very significant for trying to reason about properties of the language in general. Without a restriction for the inference process to be finite, the semantics of the language are not provable deterministic, and no computational induction can be used. The proofs of the Determinacy Theorem and of the associativity of the `append` function cannot be performed from the equational/axiomatic representation of the semantics.

Several other minor problems were encountered with the original representation:

- For Axiom 6.3, the presence of Obj_NONE on the left of the equality makes it impossible to say anything conclusive about similar sentences with different background objects. Such sentences are not allowed by the semantics, but this must be represented. The problem is easily solved by replacing Obj_NONE by *obj* and adding the conjunct $obj = \mathsf{Obj\_NONE}$ to the disjuncts on the right.

- The rules for pattern matching were incorrectly specified. The Pattern Matching Theorem mentioned in Section 2.3.2 shows that changes in state can never occur during pattern matching. However, this was *explicitly* stated with each pattern matching axiom, which does not represent the spirit of the original definition. The rules had not been expanded by either the state or exception conventions. See Section 7.2 and Appendix B.2 for a descriptions of the Pattern Matching Theorem and its proof in HOL.

6.2 A Correct Representation

6.2.1 Completing the Semantics

The previous section demonstrates some of the dangers of using an axiomatic approach for embedding objects in HOL. The use of axioms could not be avoided to make the last type definitions in Chapter 5, but in those cases it was claimed that these axioms could be derived as theorems from a proper definition of the types in HOL. In this section a representation of the semantics which does not use axioms is presented. Instead it takes a purely *definitional* approach to embedding the inference rules. This eliminates the possibility of introducing logical inconsistency, and also results in a correct representation.

The underlying problem that has been encountered is that the formal definition does not precisely specify what it means for a sentence to be valid. The intent is clear - a sentence is inferable if and only if it can be produced by the application of a finite number of the inference rules. It was noted in Section 2.2.3 that there is a one-to-one correspondence between inferable sentences and inference trees. Furthermore, it is obvious that thinking about inferability in terms of inference trees will capture the fact that computations need to be finite. This fact may be used to define inferability in the following way:

Definition 1 *A sentence is inferable if and only if there exists a valid finite inference tree whose root node is that sentence.*

Formally, an inference tree is a tree whose nodes are labeled with sentences. Only valid inference trees are of interest to us:

Definition 2 *Given an inference tree T with root node*

$$s, obj \vdash phrase \Rightarrow res, s'$$

and children $T_1, \ldots, T_n$, then T is valid if and only if its node is valid and each if its children are also valid.

An inference node N is valid if and only if, for some inference rule R, the label at the node is the conclusion of R, the premises to R are the labels of the children of N, and each side condition of R is satisfied.

In simple terms, an inference tree is valid if and only if it represents a valid computation according to the inference rules.

These two definitions complete the formal semantics for the Core. They are implicit in the definition, but the fact that they are not stated explicitly led directly to the incomplete formalisation used by Hutchins. They need to be stated explicitly in HOL.

It is interesting to note how the use of inference trees differs to previous attempts at the representation. No attempt has been made to directly represent the rules of inference as logical implications in HOL. In fact the correspondence between the semantics and the representation is quite hidden. Given this, it would seem more accurate to think of the inference rules as defining allowable constructions of finite inference trees, rather than as individual logical rules.

6.2.2 Defining Inference Trees in HOL

Given the notions introduced in the previous section, it is now possible to proceed to make the appropriate definitions in HOL. The following predicates will capture the relevant notions:

VALID_INFR :	*Sentence* → *(Sentence)list* → *bool*	≡	validity of an inference node
VALID :	*ltree* → *bool*	≡	validity of an inference tree
IS :	*Sentence* → *bool*	≡	inferability of a sentence

Validity of an inference node is defined first. This is done by by cases, according to the type of phrase used in the sentence. A simple example is for those *AtExp* phrases which are just variables. The predicate AtExp_Var_VALID is defined as follows:

2

```
⊢def  AtExp_Var_VALID var children (s,env,obj,res,s') =
    (∃val VE EE.
      (children = []) ∧
      (obj = Obj_NONE) ∧
      (res = Res_Val val) ∧
      (s' = s) ∧
      (env = Env VE EE) ∧
      (APPLY var VE = RESULT val))
```

The predicate will only be true under the following conditions: there are no sub-inferences at the node; there is no side object; the result of sentence is some value *val*; there is no state change; and the result of applying *var* to *env* is *val*. These conditions are stated in the conjuncts of the above definition. Note that no new axiom has been introduced, but rather simple definition has been made.

Such a predicate is introduced for each kind of phrase. The case for single value bindings illustrates how the children that an inference node needs are specified:

3

$$
\begin{array}{l}
\vdash_{def}\ \texttt{ValBind\_ValBind\_VALID}\ pat\ exp\ children\ (s,obj,res,s') = \\
\quad (\exists VE\ val\ s''. \\
\quad\quad (obj = \texttt{Obj\_NONE}) \land \\
\quad\quad (res = \texttt{Res\_VarEnv}\ VE) \land \\
\quad\quad (children = \\
\quad\quad\ \ [s,E,\texttt{Obj\_NONE},\texttt{Ph\_Exp}\ exp,\texttt{Res\_Val}\ val,s''; \\
\quad\quad\quad s'',E,\texttt{Obj\_Val}\ val,\texttt{Ph\_Pat}\ pat,\texttt{Res\_VarEnv}\ VE,s'])) \lor \\
\quad (\exists val\ s''. \\
\quad\quad (obj = \texttt{Obj\_NONE}) \land \\
\quad\quad (res = \texttt{Res\_Pack(Pack(ExVal\_Name(ExName `Bind`))))}) \land \\
\quad\quad (children = \\
\quad\quad\ \ [s,E,\texttt{Obj\_NONE},\texttt{Ph\_Exp}\ exp,\texttt{Res\_Val}\ val,s''; \\
\quad\quad\quad s'',E,\texttt{Obj\_Val}\ val,\texttt{Ph\_Pat}\ pat,\texttt{Res\_FAIL},s'])) \lor \\
\quad (\exists p. \\
\quad\quad (obj = \texttt{Obj\_NONE}) \land \\
\quad\quad (res = \texttt{Res\_Pack}\ p) \land \\
\quad\quad (children = [s,E,\texttt{Obj\_NONE},\texttt{Ph\_Exp}\ exp,\texttt{Res\_Pack}\ p,s']))
\end{array}
$$

Note the similarity between the right hand side of the definition and the Axiom 6.3. The IS predicates have been replaced by assertions about the children of the node in the inference tree.

Once all the definitions have been made for a particular phrase type, they are collected together to define validity for nodes of that phrase type:

4

$$
\begin{array}{l}
\vdash_{def}\ (\texttt{AtExp\_VALID (AtExp\_Var}\ var) = \texttt{AtExp\_Var\_VALID}\ var) \land \\
\quad (\texttt{AtExp\_VALID (AtExp\_Con}\ con) = \texttt{AtExp\_Con\_VALID}\ con) \land \\
\quad (\texttt{AtExp\_VALID (AtExp\_ExCon}\ excon) = \texttt{AtExp\_ExCon\_VALID}\ excon) \land \\
\quad (\texttt{AtExp\_VALID AtExp\_Record\_Empty} = \texttt{AtExp\_Record\_Empty\_VALID}) \land \\
\quad (\texttt{AtExp\_VALID (AtExp\_Record}\ record) = \texttt{AtExp\_Record\_VALID}\ record) \land \\
\quad (\texttt{AtExp\_VALID (AtExp\_Local\_Dec}\ dec\ exp) = \texttt{AtExp\_Local\_Dec\_VALID}\ dec\ exp) \land \\
\quad (\texttt{AtExp\_VALID (AtExp\_Paren}\ exp) = \texttt{AtExp\_Paren\_VALID}\ exp)
\end{array}
$$

The definitions of the phrase types are then combined to form the definition of VALID_INFR:

5

```
⊢def (VALID_INFR (Ph_AtExp atexp) = AtExp_VALID atexp) ∧
    (VALID_INFR (Ph_ExpRow exprow) = ExpRow_VALID exprow) ∧
    (VALID_INFR (Ph_Exp exp) = Exp_VALID exp) ∧
    (VALID_INFR (Ph_Match match) = Match_VALID match) ∧
    (VALID_INFR (Ph_MRule mrule) = MRule_VALID mrule) ∧
    (VALID_INFR (Ph_ValBind valbind) = ValBind_VALID valbind) ∧
    (VALID_INFR (Ph_ExBind exbind) = ExBind_VALID exbind) ∧
    (VALID_INFR (Ph_Dec dec) = Dec_VALID dec) ∧
    (VALID_INFR (Ph_AtPat atpat) = AtPat_VALID atpat) ∧
    (VALID_INFR (Ph_PatRow patrow) = PatRow_VALID patrow) ∧
    (VALID_INFR (Ph_Pat pat) = Pat_VALID pat)
```

These definitions are made using the mutually recursive types package. Essentially then, VALID_INFR represents the disjunction of all the possible inference steps. We are now in a position to define VALID for the validity of an inference tree:

6

```
⊢def  VALID (Node (s,obj,ph,res,s') ch) =
       VALID_INFR ph (MAP ROOT ch) (s,obj,res,s') ∧ EVERY VALID ch
```

Here VALID has type $(Sentence)ltree \rightarrow bool$. The type *ltree* is the predefined HOL type for labeled trees. VALID is a recursive definition over these trees. ROOT is a simple definition over the type *ltree*. The higher order functions EVERY and MAP are defined in the HOL list theory and have the usual properties.

The validity of sentences can now be defined:

7

```
⊢def IS sent = (∃tree. VALID tree ∧ (ROOT tree = sent))
```

This follows the definition that was given in Section 6.2.1. That is, a sentence is inferable if and only if it has a corresponding valid inference tree whose root is that sentence.

The definitions are now complete, apart from the few small changes we shall make in the next section. The semantics that have been defined here shall be used extensively in the following chapters.

6.2.3 Eliminating Free Choice

As mentioned in Section 2.3.3 the dynamic semantics of Standard ML are non-deterministic. This is due to the side conditions of rules 114 and 138 where new addresses and exception names are chosen. To ensure that determinism holds strictly, we alter in a minor way the manner in which these side conditions are represented in these rules. Instead of demanding

$$a \notin \mathsf{DOM}\ memory$$

for the new address, we write

$$a = \mathsf{CHOOSE\_Addr}\ memory$$

The function CHOOSE_Addr is defined using the ε-choice operator:

8

$\vdash_{def}$ `CHOOSE_Addr` $mem = \varepsilon addr.\ \neg(addr$ `IN (DOM` mem`))`

The definition ensures that the only information that is known about the new address is that it is not currently in use. However we do know one thing which is not specified in the semantics - that every time we choose a new address for a particular memory *mem* we will get the same new address. In other words address choice in now deterministic.

This new property could be dangerous if it were possible to reason about addresses themselves. For instance, a verification proof might rely on the fact that two addresses chosen in two different instances were the same. However, this should not be a problem, for the typing rules of Standard ML ensure that the operation of programs do not rely on the addresses chosen for updatable references.[1]

To ensure that the ε-choice operator may be eliminated from proofs easily, the following results must be proved:

9

$\vdash \neg$`(CHOOSE_Addr` mem`) IN (DOM` mem`)`
$\vdash$ `FINITE` $enames \supset\ \neg$`(CHOOSE_ExName` $enames$`) IN` $enames$

These rely on proofs that the set of addresses and the set of exception names are infinite, for if they were not then the choice function might fail. The HOL details of these proofs are not shown here.

6.3 The Dynamic Basis

The remaining constructs that need to represented in HOL to complete the embedding of the dynamic semantics are those associated with the initial Standard ML environment, called the *dynamic basis*. The base values in the dynamic basis were described briefly in Section 2.2.4. The semantics of the base values are defined through the function APPLY, which must be renamed to SML_APPLY to avoid a name conflict with the APPLY function in the finite partial functions theory.

The type of SML_APPLY is $BasVal \to Val \to Result$. The type $BasVal$ was defined as one of the semantic types by

$$\begin{array}{rcl} BasVal & = & \mathsf{BasVal\_plus} \\ & | & \mathsf{BasVal\_minus} \\ & \vdots & \\ & | & \mathsf{BasVal\_equal} \end{array}$$

We will define SML_APPLY in terms of a single function for each *BasVal*, e.g. *plus* for BasVal_plus. The Formal Definition's description of the arithmetic functions seems simple enough:

[1]The modification can be justified on other grounds. The addresses in the semantics do not necessarily have to respond to addresses in an implementation, but could be considered to be simply some ordering of the addresses that will be chosen in an execution of a program fragment.

> We abbreviate SML_APPLY(+,$\{1 \mapsto a, 2 \mapsto b\}$) to $a + b$ ... The arithmetic functions /,*,+,- all return the results of the usual arithmetic operations, or exception packets, respectively [Quot], [Prod], [Sum], [Diff] if the result is undefined or out of range.

Fortunately, HOL provides an arithmetic theory which defines the "usual arithmetic operations". However, it is still crucial that care be taken to organise the HOL versions of these definitions as well as possible. This representation problem is useful to demonstrate that it is important to have special representations some Standard ML values. A naïve representation of addition might be:

10

```
⊢def plus val =
     let rec = (εrec. val = Val_Record rec) in
     let n = (εnum. APPLY(Lab '1') rec = RESULT(Val_Con(Con_Num num)))
     and m = (εnum. APPLY(Lab '2') rec = RESULT(Val_Con(Con_Num num))) in
        (n + m > SML_MAXINT) → Res_Pack (Pack (ExCon 'Sum'))
                             | Res_Val (Val_Con (Con_Num (n + m)))
```

Note that "let ... = ... in ... " is a HOL logical construction, and not a programming construct.

The definition, though logically correct, is grotesque, and would be difficult to manipulate in a proof, for it involves *three* ε-choice operators and many other constructs. The ostensible problem causing the complication is the extracting argument information from the input value *val*. The wider problem is that the input is always a pair of numbers, which is a form of value which is worth representing explicitly.

This problem raises the wider question of the most appropriate way to access Standard ML values and types in HOLlogical terms. If the primitive constructors of the *Val* type are relied upon we will consistently run into the problem exhibited above. The general problem is outside the scope of this thesis. However, in Chapter 10 techniques shall be presented which solve the problem for concretely structured Standard ML types. Some theories for some important Standard ML types will also be developed, in particular for numbers, pairs, booleans and lists.

For the moment it is worth noting that with the results developed in these theories the definition of addition becomes:

11

```
⊢def plus val =
     let n = MLNUM_TO_NUM (MLPAIR_FST val)
     and m = MLNUM_TO_NUM (MLPAIR_SND val) in
       OUT_OF_RANGE(n+m) → Packet 'Sum' | Res_Val (NUM_TO_MLMLNUM (n + m))
```

Note that "MLNUM" is *not* a new HOL type, but rather represents a subset of the type *Val*. The definition of equal is:

12

```
⊢def equal val =
      let n = MLPAIR_FST val
      and m = MLPAIR_SND val in
         (m = n) → Res_Val (MLBOOL T)  Res_Val (MLBOOL F)
```

A function similar to plus or equal is defined for each *BasVal*. SML_APPLY is then defined in terms of them:

13

$\vdash_{def}$ `SML_APPLY BasVal_plus` val = `plus` val $\wedge$
`SML_APPLY BasVal_minus` val = `minus` val $\wedge$
...
`SML_APPLY BasVal_geq` val = `geq` val

The full definition of these functions can be found in appendix E.

Chapter 7

Language Properties and Proof Techniques

In the last chapter the HOL representation for the Dynamic Semantics of the Core language was completed. In one sense a complete theory has now been constructed, for *any* fact which may be logically deduced from the Core dynamic semantics can now be proven in HOL. Importantly, there is no requirement that further definitions or type specifications be made in HOL, although we are free to add definitions when it seems appropriate.

With regard to the current state of the theory that has been developed, it is useful to draw a parallel with reasoning about the natural numbers. The HOL type *num* is defined in terms of 0 and SUC. However, pragmatic considerations make it essential that the primitive theory be enriched with theorems and a whole range of new definitions. This is effectively the aim of the chapters that follow - to expand the present theory of Standard ML with theorems, rules, tactics and definitions to produce an environment which makes it easier to work with the semantics that have been defined.

In this chapter a series of fundamental properties about inferability are proved. These results will be extremely useful in simplifying the process of reasoning about the semantics. Characterising the semantics in terms of inference trees is logically correct, but difficult to work with. For example, if we now tried to prove the associativity of the append function, as outlined in Section 2.3.4, then at every step of the proof it would be necessary to refer back to the definitions of VALID, VALID_INFR and IS. This would make the proof extremely long and the essential argument would be lost among the details. The results developed in this chapter will allow reasoning to be performed about the semantics without direct reference to inference trees.

In this chapter two of the theorems described in Chapter 2 are also proved: the Pattern Matching Theorem and the Determinacy Theorem. The importance of these results in verifying the correctness of the representation is undeniable.

7.1 Backward Rewrite Rules

The first set of theorems that shall be proved correspond exactly to the axioms asserted by Hutchins in his representation of the semantics. There are 43 of these in total, one for each kind of sub-phrase. These results are referred to as "backward" rules, for they state what conditions must hold once a sentence is known to be inferable.

7.1.1 A Simple Case

For phrases of the form "`Ph_AtExp (AtExp_Var` *var*)" we want to prove the following goal:

goal 1

$$\texttt{"IS(}s,\ E,\ obj,\ \texttt{Ph\_AtExp (AtExp\_Var}\ var\texttt{)},\ res,\ s'\texttt{)} \ = \\ (\exists val\ VE\ EE. \\ (obj = \texttt{Obj\_NONE}) \wedge (res = \texttt{Res\_Val}\ val) \wedge (s' = s) \wedge \\ (E = \texttt{Env}\ VE\ EE) \wedge (\texttt{APPLY}\ var\ VE = \texttt{RESULT}\ val)\ \texttt{)"}$$

The conjuncts on the right come directly from the definition of AtExp_Var_VALID and specify exactly the conditions that need to hold to infer a sentence containing phrases of the given form.

The details of the proof as constructed in HOL are shown in Appendix B.1. The proof follows fairly easily from the definitions of IS, VALID_TREE, VALID, AtExp_VALID and AtExp_Var_VALID. To aid this and other proofs two powerful special purpose tactics `SMART_ELIMINATE_TAC` and `SMART_EXISTS_TAC` are constructed.[1]. The first eliminates variables from a proof, while the second makes informed choices for witnesses when proving existential goals. Used together these make the goal simple to solve.

7.1.2 A Complicated Case

For phrases whose evaluation may involve sub-inferences, the goal we want to prove is more complicated. For the Exp_AtExp case our goal is:

goal 2

$$\texttt{"IS(}s,E,obj,\texttt{Ph\_Exp (Exp\_AtExp}\ atexp\texttt{)},res,s'\texttt{)} = \\ (\exists val. \\ (obj = \texttt{Obj\_NONE}) \wedge \\ (res = \texttt{Res\_Val}\ val) \wedge \\ \texttt{IS(}s,E,\texttt{Obj\_NONE},\texttt{Ph\_AtExp}\ atexp,\texttt{Res\_Val}\ val,s'\texttt{))} \vee \\ (\exists p. \\ (obj = \texttt{Obj\_NONE}) \wedge \\ (res = \texttt{Res\_Pack}\ p) \wedge \\ \texttt{IS(}s,E,\texttt{Obj\_NONE},\texttt{Ph\_AtExp}\ atexp,\texttt{Res\_Pack}\ p,s'\texttt{))}$$

The goal states the two possible cases for the evaluation of an expression Exp_AtExp *atexp*. To infer any result for the expression, the atomic expression *atexp* must have evaluated to either a value or an exception. The two cases are stated in the disjuncts in the right.

The proof of this result is complicated by the presence of the disjuncts on the right. Most of the 43 axioms that need to be proved have a disjunctive component, and thus we construct a proof technique which chooses between the different disjuncts based on the information in each. When used at the appropriate stage in the proof, this tactic is capable

[1]Both of these tactics, and the *fpf* theory developed in Chapter 5, have been submitted to the HOL contributors directory for the next release of the HOL system

of choosing the correct disjunct for each case we consider. This full proof technique is also outlined in Appendix B.1.

Generalized proof techniques embodied in tactics like `SMART_EXISTS_TAC` allow us to prove all 43 goals with the one tactic - `PROVE_BWD_RULE_TAC`. This tactic is both simple and robust because it models the common logic behind each proof. Each proof is essentially similar, and HOL lets us take advantage of this by using generalised proof techniques.

7.2 The Pattern Matching Theorem

In this section the results of the previous section shall be used to prove the Pattern Matching Theorem described in Section 2.3.2. In HOL the goal is stated as:

goal 1

"IS(s, E, Obj_Val val, pat, res, s') $\supset$
  (($\exists VE.$ res = Res_VarEnv VE) $\vee$ (res = Res_FAIL)) $\wedge$ (s = s')"

On the left of the implication is the assumption that some result can be inferred for matching the value *val* against the pattern *pat*. The two facts stated by the Pattern Matching Theorem are on the right hand side of the implication. The first conjunct states that either a new set of variable bindings is produced or else an exception occurs. The second conjunct asserts that the initial and final states are the same, and thus that no change of state occurs.

The details of the HOL proof for the theorem are shown in Appendix B.2. The result is proved by structural induction over the syntax of patterns. That is, the result must be proved for each of the "base" patterns - constants, variables, wildcards and wildrows, and then for each pattern construction method as specified by the grammar for patterns. This is the first major application of the mutual recursion theorems developed in Chapter 4, for the pattern grammar is mutually recursive.

It is interesting to note that the proof is simple, although it is not because we are proving a trivial property. Rather, the backward rewrite rules from the previous section make it easy to reason inductively, and the rules eliminate the need to refer to inference trees in the proof.

This theorem has some significance for program verification, but it is useful to prove it for other reasons:

- It helps to verify the correctness of the representation of the semantics. Very often, mistakes in specifying semantics cause theorems such as this to be unprovable. In fact, such mistakes were discovered and fixed during the proof of this theorem.

- Its proof is interesting and instructive. In particular it demonstrates how the mutually recursive induction theorems may be employed.

7.3 Forward Inference Rules

The natural way to think about an inference rule is as an implicative theorem, where the premises of the rule imply the conclusion. This section proves that each of the inference rules hold in this form. The resulting theorems are called "forward inference rules", as they may be used to construct inferable sentences from sub-inferences. As with the backward rewrite rules, this set of theorems is incomplete as a total characterization of the semantics, yet each of the rules is individually true. The rules shall be used extensively in the next chapter as the basis for a symbolic evaluator for Standard ML.

There are 86 forward inference rules in total, corresponding to the inference rules in the Formal Definition. For example, one of the rules for Exp_AtExp phrases is:

goal 1

$\vdash$ `IS(`s`,`E`,Obj_NONE,`$atexp$`,Res_Val` val`,`s'`)` $\supset$
`IS(`s`,`E`,Obj_NONE,`Exp`AtExp` $atexp$`,Res_Val` val`,`s'`)`

The theorem states that if it can be inferred that the atomic expression *atexp* evaluates to the value *val* then it is possible to infer the same value for Exp_AtExp *atexp*.

These rules were originally derived by Matthew Hutchins from his axiomatic definition of the semantics. We have now proved each of his axioms as theorems in our theory, and so the proofs developed by Hutchins remain valid. However, techniques for streamlining and simplifying these proofs were developed to make their reconstruction easier should small changes occur in the semantics. The proofs follow fairly easily from the backward rewrite rules, and are described in Appendix B.3.

7.4 The Determinacy Theorem

The theorem that is proven in this section is perhaps the most important single result proved in this thesis. It is the Determinacy Theorem, outlined in Section 2.3.3. The theorem as described there would be difficult to state and prove due to the essential non-determinism of address and exception-name choosing. However, the changes made to the semantics in section 6.2.3 ensure that the choices are now deterministic, even if "unknown".

The goal for the Determinacy Theorem is quite simple to state in HOL:

goal 1

"`IS(`s`,`env`,`obj`,`ph`,`res`,`s'`)` $\wedge$ `IS(`s`,`env`,`obj`,`ph`,`res'`,`s''`)` $\supset$
    `(`$res = res'$`)` $\wedge$ `(`$s' = s''$`)`"

The goal asserts the equality of the results and final states of the two inferable sentences.

It is tempting to try to prove this goal by structural induction over the phrase *ph*. The proof hopefully progress in a similar fashion to that for the Pattern Matching Theorem. This approach almost solves the goal, but not quite. The reason why can be seen if we consider the case of phrases of the form `Ph_Exp (Exp_Appl` *exp atexp*`)`, whose evaluation is determined in part by Rule 117 of the formal semantics:

$$\frac{E \vdash exp \Rightarrow (match, E', VE) \quad E \vdash atexp \Rightarrow v \quad E' + \mathsf{Rec}\ VE, v \vdash match \Rightarrow v'}{E \vdash exp\ atexp \Rightarrow v'}$$

When structural induction is used, we are allowed to assume that our goal holds for each of the *syntactic* sub-cases in each particular step case. In the case of *exp atexp*, it is thus possible to assume that the determinacy result holds for the evaluation of both *exp* and *atexp*. Typically, *exp* will evaluate to a closure, and *atexp* a value. It is *not* possible to assume that the determinacy result holds for the third premise to the rule:

$$E' + \mathsf{Rec}\ VE, v \vdash match \Rightarrow v'$$

This sentence is the evaluation of the function body which is contained within the closure. In other words this represents the call to the function itself. Function application accesses information outside the immediate syntactic context, and so structural induction down the syntax is not sufficient.

Any proof of this theorem must use some form of *computational* induction. That is, an induction must be performed over the computation process itself. As computations have been modeled by finite inference trees, we can induct over the depth of the inference process. First the definition of the depth of a tree must be made:

1

```
MAX_DEF   = ⊢def ∀m n. MAX m n = (m < n → n | m)
MAXL_DEF  = ⊢def (MAXL [] = 0) ∧ (MAXL (CONS hd tl) = MAX (MAXL tl) h)
DEPTH_DEF = ⊢def DEPTH(Node v tl) = 1 + (MAXL(MAP DEPTH tl))
```

We now define a new version of inferability which puts a limit on the depth of inference for the sentence, using the predicate ISDB (Inferable Sentence Depth Bounded). $\mathsf{ISDB}\ n\ sent$ is defined to be true if the sentence *sent* may be inferred by an inference tree with depth $\leq n$:

2

```
ISDB_DEF    = ⊢def ISDB n sent =
        ∃tree. (ROOT tree = sent) ∧ VALID tree ∧ (DEPTH tree ≤ n)
IS_EQ_ISDB = ⊢ IS sent = (∃n. ISDB n sent)"
```

The second theorem shows the relationship between depth-bounded and general notions of inferability.

The essence of the proof of the Determinacy Theorem can now be stated in the following goal, which we shall call the Determinacy Lemma:

goal 2

```
"∀n.
    ISDB n (s,env,obj,ph,res,s') ∧
    ISDB n (s,env,obj,ph,res',s'') ⊃
          (res = res') ∧ (s' = s'')"
```

The lemma states that if two inferences both have depth $\leq n$ then they satisfy the Determinacy property.

The computational induction is now performed by inducting on n. With some rewriting this results in some 400 odd sub-cases that arise by the different ways in which the two sentences on the left of the implication could have been inferred.

The method that is used to prove each sub-goal is shown in Appendix B.4. As a preliminary to the proof, 43 backward rewrite rules about ISDB are proven. These are quite similar to the backward rewrite rules for IS. These results are proven separately because they may well turn out to be of more general use in other proofs that use computational induction. With these results the proof of all the sub-goals can be streamlined through one tactic.

The proof of the Determinacy Theorem follows from the lemma above. The lemma states that if two inferences over the same background and phrase both have depth $\leq n$ then their results are equal. The theorem follows by considering n to be the maximum depth of the two inferences on the left of the implication. Both sentences are then inferable within depth n and so the result follows. We omit the details of the HOL proof.

7.4.1 The Significance of the Theorem

The Determinacy Theorem ensures that each evaluation has at most one result. Unfortunately this result may not be strengthened to *exactly* one result, for two reasons. Firstly, as we saw in Chapter 6, there are infinite computations for which no result can be inferred. Secondly, many computations do not make sense under the typing rules of the language, and these in general have no valid inferences. For example, no result can be inferred for the evaluation of the phrase "6 x", where we try to use 6 as a function.

Determinism justifies the following definition of evaluation:

3

$$
\begin{aligned}
&\vdash_{def} \;\; \texttt{EVAL}\ (s,E,obj)\ \texttt{phrase} = \\
&\quad ((\exists res\ s'.\ \texttt{IS}(s,E,obj,phrase,res,s')) \rightarrow \\
&\quad\; \texttt{RESULT}\ (\varepsilon(res,s').\ \texttt{IS}(s,E,obj,phrase,res,s'))\ | \\
&\quad\; \texttt{FAILURE})
\end{aligned}
$$

EVAL maps a background and a phrase to either a pair containing a result and the final state, or else the special constant FAILURE, depending on whether a successful evaluation exists or not.[2] The following theorems, which capture the full relationship between evaluation and inferability, can be proven fairly easily:

4

$$
\begin{aligned}
&\vdash \texttt{IS}(s,E,obj,phrase,res,s') \supset \;(\texttt{EVAL}\ (s,E,obj)\ phrase = \texttt{RESULT}\ (res,s')) \\
&\vdash (\texttt{EVAL}\ (s,E,obj)\ phrase = \texttt{RESULT}\ (res,s')) \supset \;\texttt{IS}(s,E,obj,phrase,res,s') \\
&\vdash \texttt{IS}(s,E,obj,phrase,res,s') = (\texttt{EVAL}\ (s,E,obj)\ phrase = \texttt{RESULT}\ (res,s'))
\end{aligned}
$$

In other words, inferability implies successful evaluation, and a successful evaluation implies inferability.

[2] In this thesis we will generally omit the RESULT constructor, unless ambiguity results.

Chapter 8

Symbolic Evaluation

This chapter describes an automated method for constructing proofs about language properties known as *symbolic evaluation*. The logical basis for the evaluator is the set of forward inference rules proved in Chapter 7.

Evaluation of languages is, of course, a well understood process. For instance it would not be very difficult to implement HOL-ML program to evaluate the syntactic terms for Standard ML programs. An example of how this would work might be be:

1

```
# sml_eval("let val x = 6 in x end");;

"Val_Con (Con_Num 6)";;
```

In this example the evaluating function simply returns the appropriate value for the expression. Many details have been omitted here - the evaluator would need to be supplied with an initial state, environment and background object, all fully specified. Also, note that the Standard ML input code would need to be parsed - quotes have been used in the example above to indicate the HOL syntactic term for the parsed code. Lastly, evaluation should also produce a modified state, which would have to be returned.

This approach may help to discover the result of an evaluation, but as yet no theorem has actually proved about the evaluation. As such little has been achieved in terms of reasoning about the language. What we need is an evaluator which proves a theorem about the evaluation process, something like:

2

```
# SML_EVAL (state,env, obj) "let val x = 6 in x end";;

⊢ EVAL (state, env, obj) "let val x = 6 in x end"  =  6_Val , state";;
```

Note that in this example the end result of the evaluator is a theorem asserting the value of the EVAL function for the given computation.

Hutchins, as part of his project in 1990, implemented just such an evaluator. His program was remarkable for its functionality, clean design and implementation. The contributions made to the evaluator in this project were essentially just additions, corrections and refinements to that program. The additions were, however, substantial enough to enable the evaluator to be used for the proofs presented in Chapter 10.

8.1 Basic Symbolic Evaluation

Making the evaluator prove a result about a completely specified instance of a program may seem rather pointless, for it is rare that one wants to reason about just one instance of a program's execution. However, the evaluator can be used to reason about infinite families of computations by using *symbolic* evaluation. For example, consider the small expression `let val` *var* `=` *n* `in 1 end`. It should be possible to prove that this always evaluates to the value 1 regardless of *var* and *n*. A symbolic evaluator only needs its input to be partially specified. In the context of the HOL logic, unspecified portions of the input will be represented by variables.

Often the symbolic evaluator will come to an evaluation which it cannot compute. One technique for handling this is to split the computation into two or more paths, each of which makes a different assumption about the symbols involved. For instance, consider evaluating the match $M =$ `true => false | false => true`, which implements negation. When the match is applied to the value `true` the evaluator produces the theorem

3

$$\vdash \texttt{EVAL (}s\texttt{, }E\texttt{, true) }M = \texttt{ false, }s$$

However, if we apply an arbitrary value v to the match v then we get three different theorems:

4

$$\vdash (v = \texttt{true}) \supset \texttt{EVAL (}s\texttt{, }E\texttt{, }v\texttt{) }M = \texttt{false, }s$$
$$\vdash \neg(v = \texttt{true}) \wedge (v = \texttt{false}) \supset \texttt{EVAL (}s\texttt{, }E\texttt{, }v\texttt{) }M = \texttt{true, }s$$
$$\vdash \neg(v = \texttt{true}) \wedge \neg(v = \texttt{false}) \supset \texttt{EVAL (}s\texttt{, }E\texttt{, }v\texttt{) }M = \texttt{[Match], }s$$

The theorems correspond to the cases where the value matches first pattern, the first pattern match fails but the value matches the second, and where both matches fail. The last case causes an exception to be raised. The assumptions introduced about v differentiate between the different cases.[1]

8.1.1 Evaluating under Assumptions

The symbolic evaluator is capable of using extra information to aid it in the evaluation process. In particular, it accepts a list of theorems of the form

$$\vdash \forall v1 \; \ldots \; vn.\ \texttt{EVAL (}s\texttt{, }E\texttt{, }obj\texttt{) }phrase = res\texttt{, }s'$$

Before the evaluator attempts the evaluation of any phrase it attempts to discover an instantiation of one of these theorems gives an evaluation result for the computation being considered. For example, under the assumption

[1]Note that the evaluator knows nothing particular about the types of expressions involved in the computation, and thus can't tell that the third case will never happen.

$\forall s.$ EVAL (s, E, Obj_NONE) $exp = val$, s

the following theorem can be proven by the evaluator:

$\vdash$ EVAL (s'0, E, Obj_NONE) "let x = exp in x" = val, s'0

The main use of this facility in this project is to provide induction hypotheses to the evaluator. The set of theorems recognised as useful by the evaluator could be greatly expanded, but at present the emphasis has been on improving the actual deductions that the evaluator can perform.

8.1.2 Implementing the Evaluator

The main purpose of the evaluator is to abstract away from proofs the tiresome process of reasoning about the inference rules. As such it is not strictly necessary for the user of the evaluator to know how it works internally. However, in order to understand the extensions to the evaluator described in this and the next chapter, some understanding of its operation is needed.

Internally the evaluator performs all of its deductions in terms of inferable sentences and the IS predicate. The theorems proven about EVAL in the previous chapter assert the complete equality between inferability and evaluation, and thus the transformation of the results to results about EVAL is trivial. The deductions performed by the evaluator are based on the forward inference rules developed in the previous chapter. The design of the evaluator is covered in some detail in [12]. The theorem construction process is best described by an example of the HOL-ML code that carries out the evaluation:

```
SML_EVAL_Exp_AtExp (s,E,atexp) =
   let sub_evaluations = SML_EVAL_Ph_AtExp (s,E,atexp) in
      map eval sub_evaluations
   where eval (res,s',thm) =
      if (is_Res_Val res) then
         let val = dest_Res_Val r in
         [(res,s', MP (SPECL [s;E;atexp;val;s'] SML_Exp_AtExp_Rule1) thm)]
      else if (is_Res_Pack r) then
         let p = dest_Res_Pack res in
         [(res,s', MP (SPECL [s;E;atexp;p;s'] SML_Exp_AtExp_Rule2) thm)]
      else fail
```

The function shown constructs a theorem for the evaluation of the expression Exp_AtExp *atexp*, in the state s and environment E. The first evaluation step is to call the function `SML_EVAL_Ph_AtExp` which performs the sub-evaluation of the atomic expression *atexp* in s and E. This returns a list of results, new states and theorems about the evaluation of that phrase. Each item in the list represents one computation path discovered during the evaluation of *atexp*. The list is then piped through the theorem constructing function `eval`. This function decides which forward inference rule to apply depending on the result of the sub-evaluation. If the result was a value, then rule `SML_Exp_AtExp_Rule1` is applied, and if an exception was caused the rule `SML_Exp_AtExp_Rule2` is applied. These theorems were proven in Chapter 7. The first of these rules is:

5

$\vdash \forall s\ E\ atexp\ val\ s'.$
    $\texttt{IS}(s, E, \texttt{Ph\_AtExp}\ atexp, \texttt{Res\_Val}\ val, s') \supset$
    $\texttt{IS}(s, E, \texttt{Ph\_Exp}\ (\texttt{Exp\_AtExp}\ atexp), \texttt{Res\_Val}\ val, s')$

The theorem produced by evaluating the atomic expression will have the form

6

$\vdash \texttt{IS}(s,\ E,\ \texttt{Ph\_AtExp}\ atexp,\ \texttt{Res\_Val}\ val,\ s')$

for some s, E, *atexp*, *val* and *s'*. Note that this matches the left-hand side of the implication for the rule. We specialize the rule to the particular instance of variables using the HOL function `SPECL`, then draw the implication using the HOL *modus ponens* rule `MP`. Finally the result, state and theorem are returned as one of the computation paths.

8.2 Improvements to the Evaluator

8.2.1 Evaluating Variables over Complex Environments

For terminating computations, the evaluator eventually reaches evaluation points which require no sub-evaluations. For instance, there are no sub-inferences for the evaluation of a variable, except that a theorem must be proven which states the result of applying the current environment to the variable. Besides constants, which are easily evaluated, the most common bases cases are variable evaluations.

Variable evaluation turns out to be quite troublesome in the evaluator. The reason for this is that the environment against which the evaluation is taking place can have a great variety of forms. For instance:

E
$\texttt{Env}\ \{\texttt{x} \mapsto \texttt{6}\ \}\ EE$
$E\ \texttt{+++}\ \{\texttt{x} \mapsto \texttt{6}\}$
$\{\texttt{x} \mapsto \texttt{6}\}\ \texttt{+++}\ \{\texttt{y} \mapsto \texttt{false}\}$

are all valid environments. The original evaluator was unable to perform successful evaluations for any of these. This is acceptable for the first example, where all that can be done is to assume an arbitrary value v for the variable. However, the evaluator should be able to perform successful evaluations for each of the other environments.

The solution to this problem is simply to teach the evaluator how to evaluate variables in the complex environments like the ones above. In the case of environment extension, the theoretical basis is provided by proving two rules which the evaluator will use as the basis for its reasoning:

7

$\vdash$ IS(s, E_{ext}, Ph_AtExp var, Res_Val val, s) $\supset$
IS(s, E_{orig} +++ E_{ext}, Ph_AtExp var, Res_Val val, s)

$\vdash$ (E_{ext} = Env VE EE) $\wedge$ FAILS(APPLY var VE) $\wedge$
IS(s, E_{orig}, Ph_AtExp var, Res_Val val, s) $\supset$
IS(s, E_{orig} +++ E_{ext}, Ph_AtExp var, Res_Val val, s)

The rules deal with the evaluation of a variable over the environment E_{orig} +++ E_{ext}. The first rule deals with the case when the variable is found in E_{ext} - the major half of the environment. The second rule is for the other case - when the variable is explicitly not found in E_{ext} but is in E_{orig}.

From these rules a new version of SML_EVAL_AtExp_Var, the function responsible for evaluating atomic expressions which are variables, may be implemented. This is a substantial task, for functions are also needed to automatically prove theorems about constructs such as APPLY, EXT and EXTBY. The theorems which form the basis for these functions come from the fpf theory developed in Chapter 5.

After doing this, the evaluator will prove theorems such as:

8

$\vdash$ EVAL (s, E +++ $\{$x $\mapsto$ $val\}$) (AtExp_Var x) = val, s

Arbitrarily complex nestings of environments can be solved.

8.2.2 More Improvements

Some other substantial improvements were made to the evaluator during the course of this project:

- Exceptions were added to the set of evaluable constructs.
- Most of the other language constructs that had been omitted in Hutchins' implementation of the evaluator were also implemented.
- The automatic simplification of environments is now performed. This is in addition to the extra functionality provided by evaluating variables in complex environments.
- Recursive environments are now unrolled at appropriate times.
- The original evaluator added a plethora of proof obligations, mostly involving inequalities, to the theorems produced. For instance, the simple inequality

  $\neg$(Lab ‘1‘ = Lab ‘2‘)

  would often be assumed instead of automatically proven. The automatic proof of these theorems greatly aids the usability of the evaluator.
- The evaluator automatically proves results for the application of the *BasVal*s BasVal_equal and BasVal_neq. There is also scope to add evaluator functions for the other base values as they are needed.

- An accurate trace of the evaluation process can be displayed, allowing evaluation failures to be tracked down and fixed by either correcting the goal, supplying more information or adding more functionality to the evaluator.

- The symbolic evaluator originally came to some 1500 lines of code, most of which was mutually recursive. Hence if a bug was discovered in the implementation the entire evaluator had do be reloaded. A lot of time was spent discovering the best way to implement a system which would be easy to debug, and eventually updatable assignments were used to store the evaluation functions. This turns out to be an excellent Standard ML debugging technique.

8.3 An Evaluation Tactic

It is now possible to define a tactic which automates the application of the symbolic evaluator to the proof of a goal. Many different tactics could be based on the code in the evaluator, but one fairly simple tactic was sufficient for this thesis. It will act on a goal of the form

$$\texttt{EVAL}\ (s,\ E,\ obj)\ phrase = res,\ s'.$$

The evaluation that is attempted is determined by s, E, *obj* and *phrase*. The evaluator does *not* act in a goal-directed fashion, and thus *res* and *s'* are ignored until the results of the evaluation have been discovered.

An outline of the code for EVAL_TAC is shown below:

```
let EVAL_TAC (asms,gl) =
   let helpful_theorems = map ASSUME (filter (is_EVAL o strip_forall) hyp) in
   let (s,env,obj,ph,res,s') = dest_EVAL gl in
   let results = SML_EVAL (s,env,obj) ph helpful_theorems in
   if (null results) then
      failwith `evaluation did not prove anything`
   else
      EVERY (map STRIP_ASSUME_TAC results) (asms,gl);;
```

This tactic firstly looks through the current assumptions to the goal for useful results to aid the evaluation process. Useful assumptions are those which assert evaluation results (possibly over a number of universally quantified variables). The results from the evaluation are added to enrich the assumptions to the goal.

EVAL_TAC is generally able to solve goals in one step provided that a sufficient set of assumptions are available and that they are in the right form. One example of a construct which proves particularly troublesome when trying to arrange the assumptions is the following. An assumption may contain a value pair $\{1 \mapsto val_1, 2 \mapsto val_2\}$. However, the evaluator needs a theorem with the order of extension reversed. Although the functions are provably equivalent (see Chapter 6), the evaluator doesn't know this, and thus is unable to prove the result. It is too expensive to have the evaluator automatically rearrange finite partial functions into every possible extension order, and so it is really just up to the user of the evaluator to supply theorems in the correct form. This problem is no doubt only one instance of a wider variety of problems that will occur when more extensive use is made of the evaluator. An efficient solution, probably involving automatic normalisation, needs to be investigated.

The evaluator code can also form the basis for a whole range of rules, conversions and tactics which could be useful in more general proofs. For instance, the code to prove the result of applying a finite partial function to a value has wider application.

Chapter 9

Reasoning with Standard ML Types

This chapter presents some techniques which can be used to cleanly reason about concrete Standard ML types in the context of the theory that has been constructed in Chapters 4, 5 and 6.

The need to represent information about Standard ML types has already been encountered twice in this thesis. Firstly, in the discussion of the properties of the `append` function in Chapter 2, we need to state that the arguments to function were always "list"-values. Secondly, in Chapter 6 type-related value destructors and constructors were needed to define the Standard ML base values cleanly. The proofs presented in Chapter 10 will also emphasise the need for some special way of talking about types and their values. However, the biggest motivation for the constructs presented in this chapter has been practical experience, especially after many early attempts at verifying Standard ML programs were frustrated by the sheer complexity of the value information displayed on the screen.

Values are, of course, already present in the theory, in terms of the semantic type *Val*. For instance, a pair is represented by the value `Val_Record` $\{1 \mapsto \mathit{left}, 2 \mapsto \mathit{right}\}$. Here both *left* and *right* are themselves values. The problem is essentially a pragmatic one, as the values which correspond to the types do actually exist in the theory, but no organised way of talking about them has been developed.

9.1 Concrete Types as Subsets of *Val*

When typing is considered in this section, we are not considering either the type inferencing rules of Standard ML as defined in the static semantics of the Core. In addition, the types of higher order function-values are beyond the bounds of this thesis and shall not be considered either. Essentially then we are concerned with concrete Standard ML types, that is those which can be introduced by the `datatype` declaration of the language. The set of possible values for each such type corresponds to a particular subset of the semantic type *Val*, as shown for the pair example above. This shall be referred to as the *Val*=-subset for the type. A table of some simple types and their corresponding *Val*-subsets in Figure 9.1.

Note the mechanism we use to represent polymorphism - the *Val*-subset of paired

Standard ML Type	HOL Designation	Corresponding *Vals*
`int`	MLNUM	`"Val_Con (Con_Num` n`)"`
`'a * 'b`	MLPAIR	`"Val_Record` $\{1 \mapsto L, 2 \mapsto R\}$`"`
`bool`	MLBOOL	`"Val_Con (Con_Ide ‘true‘)"` `"Val_Con (Con_Ide ‘false‘)"`
`'a list`	MLLIST	`"Val_Con (Con_Ide ‘nil‘)"` `"Val_Con_Val (Con_Ide ‘cons‘)` *mlpair*`"`

Table 9.1: Some Standard ML types and their *Val*-subsets

values includes *all* pairs regardless of the particular sub-types. It is impossible to explicitly represent the type scheme `'a * 'a` without a mechanism to ensure that the substitutions for the type variable `'a` are consistent. A solution to this problem should be part of a more general solution to the problem of typing. As such no representation of type schemes or type substitution is attempted.

Rather than trying to reason about *Val*-subsets in an adhoc fashion, it is far better to define the subsets formally and then prove basic theorems about their structure. This was one of the fundamental lessons of this project: it is only possible to reason successfully about Standard ML programs with a well structured way of talking about types. Certainly a better way is needed than constantly referring back to the primitive constructors defined in the Formal Definition for the *Val* type. The following sections present two non-exclusive techniques for capturing the structure of these subsets in HOL.

9.2 Technique 1: Mapping to an Isomorphic HOL Type

The first approach is based on the recognition that many Standard ML types are isomorphic to some existing HOL type. Indeed all types which may be introduced by the `datatype` declaration can have corresponding type definitions made in HOL using the structural type definition packages discussed in Chapter 4. For instance, the *Val*-subset for the Standard ML numbers is, roughly speaking, isomorphic to the HOL type `num`.[1] Similarly, the values for booleans are isomorphic to *bool*.

It would obviously be extremely useful to be able to transform the results already present in the HOL logic concerning types such as *num* and *bool* into theorems about the isomorphic *Val*-subsets. To do this we need to define bijections between the HOL type and the *Val*-subset, as shown in Figure 9.1. Another motivation behind this technique is to allow reasoning to be performed about the Standard ML type at the level of the corresponding HOL type, rather than in the midst of a *Val*-subset. The properties of the HOL type are then "lifted" to the *Val*-subset via the bijections between the types.

For example, to construct a mapping between the *Val*-subset for Standard ML numbers and the type *num*, the following functions need to be defined in HOL.

- IS_MLNUM: *Val* $\rightarrow$ *bool*, for testing membership of the subset.

[1] In Chapter 4 we restricted the range of Standard ML numbers to a non-negative integers SML_MININT...SML_MAXINT. For the purposes of this discussion, we may consider the range to be the entire set of natural numbers.

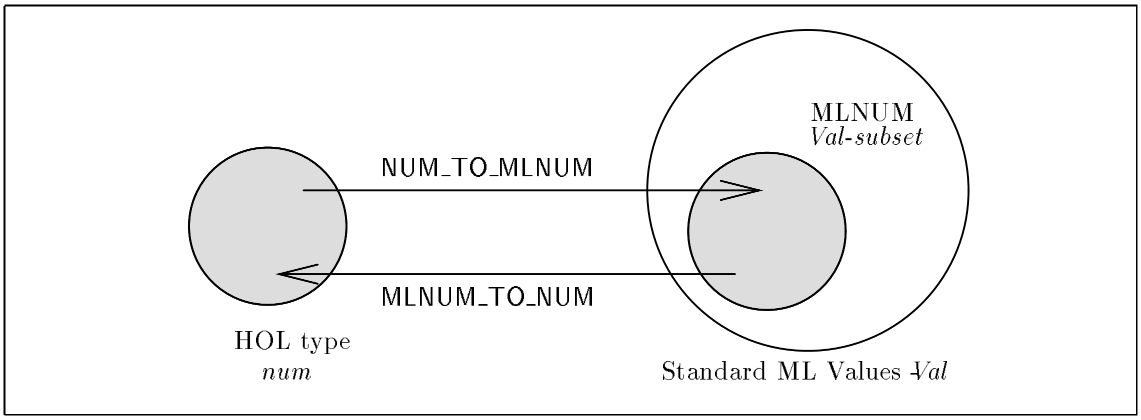

Figure 9.1: Representing Standard ML Types via an isomorphic HOL type

- NUM_TO_MLNUM: $num \rightarrow Val$, for abstraction.
- MLNUM_TO_NUM: $Val \rightarrow num$, for representation.

The definitions of these functions are fairly simple, mainly because the structure of the particular *Val*-subset is quite simple. It is fairly easy to prove the following theorems about the bijections:

```
                                                                    1
MLNUM_DEF  = ⊢def NUM_TO_MLNUM n = Val_Con(Con_Num n)
MLNUM_11   = ⊢ (NUM_TO_MLNUM n = NUM_TO_MLNUM n') = (n = n')
MLNUM_ID_1 = ⊢ MLNUM_TO_NUM (NUM_TO_MLNUM n) = n
MLNUM_ID_2 = ⊢ IS_MLNUM mlnum ⊃
                    (NUM_TO_MLNUM (MLNUM_TO_NUM mlnum) = mlnum)
```

The last two theorems capture the basic properties of the bijections between the types. These the fundamental set of theorems that need to be proven about Standard ML numbers. For instance, it is not strictly necessary to prove an induction principle over the *Val*-subset, for the bijections allow us to refer back to the induction principle for the isomorphic HOL type.

The relevant bijection theorems for Standard ML boolean values are shown below:

```
                                                                    2
MLBOOL_ID_1 = ⊢ MLBOOL_TO_BOOL(BOOL_TO_MLBOOL b) = b;
MLBOOL_ID_2 = ⊢ IS_MLBOOL val ⊃ (BOOL_TO_MLBOOL(MLBOOL_TO_BOOL val) = val)
```

9.2.1 Automatic Definition and Theorem Lifting

The benefit of defining bijections between a HOL type and a *Val*-subset is not only the fact that results from the HOL type can now be used, but also the ease with which they can be used. For instance, addition can be defined over Standard ML values as follows:

3

$\vdash_{def}$ *mlnum* `MLNUM_ADD` *mlnum′* =
    `NUM_TO_MLNUM((MLNUM_TO_NUM` *mlnum*`) + (MLNUM_TO_NUM` *mlnum′*`))`

Note what has been done to make this definition. To form the left hand side we have taken "$n + n'$", an ordinary application of the + function, and replaced each number with a member of the *Val*-subset. The right hand side has been formed similarly by employing the bijections to reduce the values to their corresponding *num*s, perform the addition and then lift the result via the NUM_TO_MLNUM mapping. This definition is straightforward in the sense that it employs the bijections MLNUM_TO_NUM and NUM_TO_MLNUM in a regular fashion. From the definition and the properties of the bijections theorems about addition can now be lifted. For instance, the theorem

4

$\vdash_{def}$ `IS_MLNUM` *mlnum* $\wedge$ `IS_MLNUM` *mlnum′* $\supset$
  mlnum `MLNUM_ADD` *mlnum′* = *mlnum′* `MLNUM_ADD` *mlnum*

is the lifted version of the addition commutativity theorem

5

$\vdash$ `n + n′ = n′ + n`

Though the process of lifting definitions and theorems is quite easy for single examples, it is a tiresome process to do manually. One of the late innovations of this project was to implement a *definition lifter* and a *theorem lifter*, both of which are capable of performing their respective operations automatically. Both mechanisms are somewhat preliminary in nature, but work correctly on a surprisingly large set of theorems. The session below illustrates an application of each lifter, firstly to make the definition of "less than" over the Standard ML number *Val*-subset, and then to prove transitivity of this function. Note particularly how the isomorphic assumptions are automatically added to the theorem produced. The analysis and theorem deduction involved is not trivial, but follows regular patterns. Some further examples of theorem and definition lifting will be shown in the Chapter 10.

6

```
# lift_definition "n < m";;
⊢def n MLNUM_LT m = (MLNUM_TO_NUM n) < (MLNUM_TO_NUM m)

# LESS_TRANS;;
LESS_TRANS = ⊢ n < n′ ∧ n′ < n″ ⊃  n < n″

# lift_theorem LESS_TRANS;;
⊢ IS_MLNUM mlnum ∧ IS_MLNUM mlnum′ ∧  IS_MLNUM mlnum″ ⊃
   (mlnum MLNUM_LT mlnum′ ∧ mlnum′ MLNUM_LT mlnum″ ⊃
      mlnum MLNUM_LT mlnum″)
```

9.3 Technique 2: General Type Definitions

The process of defining concrete subsets of the *Val* type naturally shares many features in common with the process of introducing new concrete types into HOL itself. The principal theoretical constructs that lie behind these definitions - i.e. primitive recursive functions and induction theorems, were discussed in Chapter 4. The idea behind our second technique is to derive primitive recursion and induction theorems for the subset as special cases of the structure theorems for the entire *Val* type.

The example that shall be given is for the definition of Standard ML lists. This is the first recursive type considered in this section. Lists can be introduced into Standard ML by the following datatype declaration:

```
datatype 'a list = nil | cons of ('a list)
```

This introduces new value constructors `nil` and `cons`. The first step is to define similar HOL constructors which act over objects in the *Val*-subset:

7

$\vdash_{def}$ `MLLIST_NIL = Val_Con (Con_Ide ‘nil‘)`
$\vdash_{def}$ `MLLIST_CONS` *hd tl* `= Val_Con_Val (Con_Ide ‘cons‘)`
`(Val_Record (EXT (Lab ‘1‘,`*hd*`) (EXT (Lab ‘2‘,`*tlval*`) ZIP)))`

The full definition of MLLIST_CONS is included to indicate the complexity of the value expressions that are being isolated and abstracted away by the above definitions. Once the theory is constructed, we will not have to reason with complex value constructs such as the ones shown, but rather can reason in terms of the two constructors defined above.

The predicate IS_MLLIST, which tests for membership of the subset, may now be defined. It is defined inductively using the following two rules:

8

$\vdash_{def}$ `IS_MLLIST MLLIST_NIL`
$\vdash_{def}$ `IS_MLLIST` *tl* $\supset$ `IS_MLLIST (MLLIST_CONS` *hd tl*`)`

In other words, the subset of values determined by IS_MLLIST is the smallest subset which contains the empty list and which is closed under construction. This gives us an induction principle over the *Val*-subset:

9

$\vdash$ P `MLLIST_NIL` $\wedge$
($\forall h\ tl.$ `IS_MLLIST` $tl \wedge P\ tl \supset P$ `(MLLIST_CONS` $h\ tl$`))` $\supset$
($\forall val.$ `IS_MLLIST` $val \supset P\ val$)

The result of the induction theorem is to prove P over the subset given by IS_MLLIST. Note that in the step case of the induction theorem we are allowed to assume that the value *tl* is *Val*-subset.

After a lengthy proof the following primitive recursion theorem may be deduced for the subset. The result is derived from the primitive recursion theorem for the semantic types developed in Chapter 4. The recursion theorem is specialized first to the special case of the *Val* type and then to the particular subset of this type. There is a strong similarity between this theorem and the primitive recursion theorem for normal HOL lists, which is shown below.

10

$$
\begin{array}{l}
\texttt{MLLIST\_PRIM\_REC} = \\
\quad \vdash \forall spec_{nil}\ spec_{cons}. \\
\qquad \exists fn. \\
\qquad\quad (fn\ \texttt{MLLIST\_NIL} = spec_{nil}) \wedge \\
\qquad\quad (\forall h\ t.\ \texttt{IS\_MLLIST}\ tl \supset (fn\ (\texttt{MLLIST\_CONS}\ h\ t) = spec_{cons}\ (fn\ t)\ h\ t)) \\
\texttt{list\_PRIM\_REC} = \\
\quad \vdash \forall spec_{nil}\ spec_{cons}. \\
\qquad \exists fn. \\
\qquad\quad (fn\ \texttt{NIL} = spec_{nil}) \wedge \\
\qquad\quad (\forall h\ t.\ fn\ (\texttt{CONS}\ h\ t) = spec_{cons}\ (fn\ t)\ h\ t)
\end{array}
$$

The theorem for the *Val*-subset adds the additional assertion that the primitive recursive function is only properly defined for values which satisfy IS_MLLIST.

Between the induction theorem and the primitive recursion theorem the structure of the *Val*-subset has been fully captured. The advantages of deriving these theorems is enormous. It means that new definitions over the MLLIST *Val*-subset can be made easily by specializing the primitive recursion theorem to the appropriate cases. It also makes the process of defining bijections between the HOL *list* type and the *Val*-subset quite trivial, for the appropriate recursive definitions are justified by the primitive recursion theorem. The bijection theorems:

11

$$
\begin{array}{ll}
\texttt{MLLIST\_ID\_1} = & \vdash \texttt{MLLIST\_TO\_LIST(LIST\_TO\_MLLIST}\ list) = list; \\
\texttt{MLLIST\_ID\_2} = & \vdash \texttt{IS\_MLLIST}\ val \supset\ (\texttt{LIST\_TO\_MLLIST (MLLIST\_TO\_LIST}\ val) = val)
\end{array}
$$

Some types are sufficiently simple that it may not be worth using either technique presented. The MLPAIR case is a good example. There is little need to make reference to the HOL type *prod*, for there are few interesting results stored in HOL about pair types anyway. Indeed, primitive recursion and induction theorems are not needed either, for the type is not recursive. Instead the following theorems are sufficient to characterize the *Val*-subset:

12

$$
\begin{array}{lcl}
\texttt{MLPAIR\_DEF} & = & \vdash_{def} \texttt{MLPAIR}\ l\ r = \texttt{Val\_Record(EXT(Lab `1`,}l\texttt{)(EXT(Lab `2`,}r\texttt{)ZIP))} \\
\texttt{MLPAIR\_FST} & = & \vdash \texttt{MLPAIR\_FST (MLPAIR}\ l\ r) = l \\
\texttt{MLPAIR\_SND} & = & \vdash \texttt{MLPAIR\_SND (MLPAIR}\ l\ r) = r \\
\texttt{IS\_MLPAIR} & = & \vdash_{def} \texttt{IS\_MLPAIR}\ p = (\exists l\ r.\ p = \texttt{MLPAIR}\ l\ r) \\
\texttt{MLPAIR\_11} & = & \vdash (\texttt{MLPAIR}\ l\ r = \texttt{MLPAIR}\ l'\ r') = (l = l') \wedge (r = r')
\end{array}
$$

Other results could be proved as needed.

Note that the first and second techniques that have been presented in this chapter are not mutually exclusive. The first technique shows how results from existing HOL theories can be lifted to the level of *Val*-subsets, once bijections between the HOL type and the subset have been set up. The second technique demonstrates how the primitive recursive properties of the appropriate subset of *Val* can be derived, thereby fully capturing the structural properties of the subset. This often will enable the construction of the bijections in the first place, as well as allowing reasoning about the values without reference an underlying HOL type if we so desire.

9.3.1 Towards Automation

As with the first technique, the second is quite well suited to automation. The basic methods of deducing primitive recursion functions are already available, having been developed by Tom Melham and Claudio Russo in their type definition packages. These packages need modification before they may be used to construct structured subsets, and the changes needed would be lengthy, probably several weeks work. However the flexibility and speed gained by being able to construct theories of Standard ML concrete types by merely specifying the relevant `datatype` declaration would be enormous.

Chapter 10

Verifying Standard ML Programs and Beyond

Sections 10.1 and 10.2 of this chapter describe some preliminary results regarding the application of the work of Chapters 7 and 8 to program verification. Section 10.3 discusses how some other properties of computation besides correctness, including termination. In the last section a proof of the associativity theorem for the `append` function is presented.

10.1 The Append Function

In this section we present a proof of correctness for the `append` function. The proof makes extensive use of both symbolic evaluation and the theory of Standard ML types developed in the previous section.

10.1.1 Stating Correctness

In the syntax of the Core, the append function is defined as shown below. The braced expressions on the right represent paired values in the Core syntax.

```
val rec append = fn
   {1=nil, 2=M} => M |
   {1=cons {1=x,2=L},2= M} => cons {1=x, {1=append {1=L,2=M}};
```

It is most convenient to state the correctness of a function in terms of its *match*. This is essentially the body of the function, and is stored in the closure which the function's identifier is bound to. The underlined portion above indicates the match M_{append} for `append`.

The idea behind stating the correctness of a function such as `append` is simple: we simply require that the function does indeed return the appending of its two arguments:

Statement of Correctness 1 (The Append Function) *The* `append` *function is correct if the application of* M_{append} *to* (v_1, v_2) *evaluates to* v_1 *appended to* v_2 *for all list-values* v_1 *and* v_2*. The evaluation is carried out in the recursive-environment as defined by the closure that* `append` *is bound to. There must be no change of state due to the evaluation.*

At this point the reader may well ask whether anything substantial has actually been gained by stating correctness in this way. It may seem that we are stating the meaning of one append function in terms of another. However, we have actually progressed much further than that. The HOL APPEND function is part of a well used and tested theory defined from a sound logical basis. In contrast, the Standard ML `append` function is essentially just a syntax tree, with a semantic meaning given by the Formal Definition. There is no *a priori* guarantee of its meaning apart from our own intuitions about the language. As such the theorem above is quite a strong statement, and is by no means trivial.

10.1.2 The Proof in HOL

The meaning of "append" over values is defined by lifting the definition of the HOL constant APPEND:

1

```
# lift_definition "APPEND m n";;
⊢def  MLLIST_APPEND val val′ =
        LIST_TO_MLLIST (APPEND (MLLIST_TO_LIST val) (MLLIST_TO_LIST val′))
```

The definition of correctness as given above is stated in HOL as the following goal:

goal 1

```
"∀v₁ v₂.
   IS_MLLIST v₁ ∧ IS_MLLIST v₂ ⊃
   EVAL (s, E_append, MLPAIR v₁ v₂) M_append = MLLIST_APPEND v₁ v₂, s"
```

The correctness theorem is proved by induction on the list value v_1. Because the assumptions to the theorem state that v_1 is in the list *Val*-list, we can use the induction theorem for that subset of the *Val* type. This gives two cases in which v_1 has value MLLIST_NIL and value MLLIST_CONS *hd tl*.

The Base Case

The goal for the base case is:

goal 2

```
"IS_MLLIST v₂ ⊃
  EVAL(s, E_append, MLPAIR MLLIST_NIL v₂) M_append =
                              MLLIST_APPEND MLLIST_NIL v₂, s"
```

If we now expand MLPAIR and MLLIST_NIL into their primitive values the symbolic evaluation tactic `EVAL_TAC` can be applied. The evaluation that will be performed is given on the left. The evaluator will prove that the input values match with the first match rule of the `append` function. The theorem deduced by the evaluator is:

goal 3

```
⊢ EVAL (s, E_append, MLPAIR MLLIST_NIL v₂) M_append = (v₂, s)"
```

Thus applying `EVAL_TAC` to the goal reduces it to:

2

"IS_MLLIST $v \supset$ MLLIST_APPEND MLLIST_NIL v = v"

This result is proven by automatically lifting the corresponding result for HOL lists:

3

```
APPEND_NIL = ⊢ ∀m. APPEND NIL m = m

# let MLLIST_APPEND_NIL = lift_theorem APPEND_NIL;;
MLLIST_APPEND_NIL = ⊢ ∀v. IS_MLLIST v ⊃ MLLIST_APPEND MLLIST_NIL v = v

# REWRITE_TAC [MLLIST_APPEND_NIL];;
goal proved...
```

The Step Case

The goal for the step case is:

goal 4

IS_MLLIST $v_2 \supset$
EVAL (s, E_{append}, MLPAIR (MLLIST_CONS hd tl) v_2) M_{append} =
MLLIST_APPEND (MLLIST_CONS hd tl) v_2, s

where we may assume:

4

EVAL (s, E_{append}, MLPAIR tl v_2) M_{append} = MLLIST_APPEND tl v_2, s

We again use symbolic evaluation on this goal via EVAL_TAC. The symbolic evaluator computes through the match rules of the append function, proving that the first fails. The second match rule succeeds, and the evaluator must now determine the effect of the recursive call to append. The evaluator uses the induction assumption to compute the value for this call. The resulting theorem reduces the goal a property of MLLIST_APPEND which is easily proved, again by automatically lifting a corresponding HOL *list* result.

This completes the proof for correctness of the append function. Perhaps the most remarkable thing about the proof is that it makes almost no reference to the details of the semantics of the language. The symbolic evaluator performs all the reasoning steps about the language constructs themselves that are needed. The evaluator has applied approximately 100 forward inference rules and many more results about finite maps and comparison equalities. The automated proof took five minutes, a proof by hand either on paper or in HOL would take many hours.

No direct reference has been made to the original semantic constructs developed in Chapters 4 and 6. However, the theorem ultimately rests on the representations developed there. It is just that sufficiently powerful theorems and proof tools have been constructed to allow the proof to be done fairly easily. In this proof the only real input required by the user is the specification of correctness, and the application of the appropriate induction tactic at one point.

10.2 A Small Interpreter

In this section the proof of correctness for a Standard ML interpreter for a small language is presented. The aim of the section is to demonstrate that it is practical to verify interpreters using the theory of Standard ML that has been constructed.

10.2.1 The Language and its Semantics

The language that shall be verified in this section is very small, containing only integer and boolean expressions with no imperative features. The abstract syntax for the language is:

$$
\begin{array}{lcl}
\mathit{IExp} & ::= & \underline{\mathtt{if}}\ \mathit{BExp}\ \underline{\mathtt{then}}\ \mathit{IExp}\ \underline{\mathtt{else}}\ \mathit{IExp} \\
 & | & \mathit{num} \\
\mathit{BExp} & ::= & \mathit{IExp}\ \mathtt{=}\ \mathit{IExp} \\
 & | & \mathit{bool}
\end{array}
$$

The semantics for the language are defined in a denotational style:

$$
\begin{array}{rcll}
\mathcal{M}_{\mathtt{IntExp}}(\underline{\mathtt{if}}\ b\ \underline{\mathtt{then}}\ e_1\ \underline{\mathtt{else}}\ e_2) & = & \mathcal{M}_{\mathtt{IntExp}}(e_1) & \text{if } \mathcal{M}_{\mathtt{BoolExp}}(b) \\
 & & \mathcal{M}_{\mathtt{IntExp}}(e_2) & \text{if } \neg\mathcal{M}_{\mathtt{BoolExp}}(b) \\
\mathcal{M}_{\mathtt{IntExp}}(n) & = & n & \\
\mathcal{M}_{\mathtt{BoolExp}}(e_1\ \mathtt{=}\ e_2) & = & (\mathcal{M}_{\mathtt{IntExp}}(e_1) = \mathcal{M}_{\mathtt{IntExp}}(e_2)) & \\
\mathcal{M}_{\mathtt{BoolExp}}(b) & = & b &
\end{array}
$$

10.2.2 An Interpreter for the Language

A Standard ML implementation of an interpreter for the language is shown below.[1] It is written in the syntax of the Core. However, paired values are written as `(x,y)` instead of `{1= x, 2= y}`.

```
datatype IExp = Num num
              | If  (BExp,(IExp,IExp))
     and BExp = Eq  (IExp,IExp)
              | Bool bool;

val choose =
    fn (true,  (i1,_)) => i1 |
       (false, (_,i2)) => i2;

val rec interp_iexp =  fn
           If (bexp, (iexp1,iexp2)) =>
              let path = choose (interp_bexp bexp, (iexp1, iexp2)) in
                 interp_iexp path  |
           Num n => n

    and interp_bexp = fn
           Eq (iexp1,iexp2) =>
              equal (interp_iexp iexp1, interp_iexp iexp2) |
           Bool b => b;
```

[1]The term “evaluator” would perhaps be a better description of the program, but “interpreter” is used to avoid confusion with the symbolic evaluator of Chapter 8.

It is worth noting the following points about this implementation:

- The datatype defined to represent programs is mutually recursive. The declaration introduces new constructors `Num`, `If`, `Eq` and `Bool`. Note that the declaration makes use of the predefined Standard ML types `bool` and `num`.
- The Core does *not* have conditional expressions built in, but rather uses pattern matching to discriminate between arguments. The function `choose` is used to implement a simple if-then-else choice construct between two input arguments.
- The interpretation functions make extensive use of pattern matching to determine the content their arguments and analyse the program fragments.
- The interpretation functions are mutually recursive.
- `interp_bexp` uses the Standard ML base value `equal`. In contrast the `append` function made no use of any external declarations.

10.2.3 Stating Correctness

In a similar fashion to the `append` function, M_{interp_iexp} and M_{interp_bexp} are defined to be the bodies of their respective functions. We also define E_{interp_iexp} and E_{interp_bexp} to be the environments that are stored with the closures of these functions after their declarations. By the semantics of mutually recursive declarations it can be shown that E_{interp_iexp} and E_{interp_bexp} will be the same environment, which we shall call E.

The correctness of the interpreter is stated in the following way:

Statement of Correctness 2 (The Interpreter) *The interpreter is correct if*

- *the application of M_{interp_iexp} to any value iexp results in no state changes and a result-value v which corresponds to the value of $\mathcal{M}_{\texttt{IntExp}}(iexp)$. It can be assumed that iexp is a Standard ML value of type* `IExp`.
- *the application of M_{interp_bexp} to any value bexp results in no state changes and a result-value v which corresponds to the value of $\mathcal{M}_{\texttt{BoolExp}}(bexp)$. It can be assumed that bexp is a Standard ML value of type* `BExp`.

Both evaluations occur in the environment E, which can be assumed to have the correct bindings for `equal`*,* `choose`*,* `interpret_iexp` *and* `interpret_bexp`.

10.2.4 Representing the Types in HOL

The first step of the proof is to represent the *Val*-subsets of the types introduced by the `datatype` declaration. The techniques of the previous chapter are used to do this. The mutual recursiveness of the types makes this process slightly more complicated, for two *Val*-subsets must be constructed simultaneously. Only a brief outline of the details will be given here. The following predicates define the subsets:

IS_MLEXP :	$(Val + Val) \rightarrow bool$	$\equiv$	a relation defining the two subsets of Val
IS_MLIEXP :	$Val \rightarrow bool$	$\equiv$	selects the first subset from IS_MLEXP
IS_MLBEXP :	$Val \rightarrow bool$	$\equiv$	selects the second subset from IS_MLEXP

IS_MLEXP is defined inductively by the construction rules of the types. IS_MLIEXP and IS_MLBEXP are defined in terms of ML_EXP. We also define *Val* constructors MLNum, MLIf, MLEq and MLBool, which correspond to the constructors of the `datatype`. In comparison the representation of Standard ML lists had just two constructors: MLLIST_NIL and MLLIST_CONS.

From these definitions the relevant induction and existence theorems which capture the structure of the two *Val*subsets can be derived. This is by far the most lengthy part of the whole proof, but as the HOL derivations are fairly straight forward the details will not be shown here. As noted in the Chapter 9, the process could be automated, and much of the automation work has already been completed.

The aim now is to set up bijections between the *Val*-subsets that have been created and a pair of underlying HOL types. There are no corresponding types pre-defined in the HOL system, but it is quite simple to introduce a new pair of mutually recursive concrete types. This is done using the mutually recursive types package mentioned in Chapter 4. The HOL types are *iexp* and *bexp*, with constructors NUM, BOOL, IF and EQ. The bijections are named in the usual fashion: IEXP_TO_MLIEXP, MLIEXP_TO_IEXP, BEXP_TO_MLBEXP and MLBEXP_TO_BEXP. After some lengthy proofs the following theorems about the bijections can be deduced:[2]

5

$$\begin{array}{ll}
\texttt{MLIEXP_ID_THM_1} = & \vdash \texttt{MLIEXP_TO_IEXP (IEXP_TO_MLIEXP } iexp\texttt{)} = iexp \\
\texttt{MLBEXP_ID_THM_1} = & \vdash \texttt{MLBEXP_TO_BEXP (BEXP_TO_MLBEXP } bexp\texttt{)} = bexp \\
\texttt{MLIEXP_ID_THM_2} = & \vdash \texttt{IS_MLIEXP } mliexp \supset \\
& \quad \texttt{IEXP_TO_MLIEXP (MLIEXP_TO_IEXP } mliexp\texttt{)} = mliexp \\
\texttt{MLBEXP_ID_THM_2} = & \vdash \texttt{IS_MLBEXP } mlbexp \supset \\
& \quad \texttt{BEXP_TO_MLBEXP (MLBEXP_TO_BEXP } mlbexp\texttt{)} = mlbexp
\end{array}$$

10.2.5 The Proof in HOL

Now that the type bijections between the underlying HOL types and the appropriate *Val*-subsets have been constructed, it is possible to construct the semantics of the language and the statement of correctness in HOL. The semantic meaning functions $\mathcal{M}_{\texttt{IntExp}}$ and $\mathcal{M}_{\texttt{BoolExp}}$ are defined without reference to the bijections, over the types *Iexp* and *Bexp*:[3]

6

$$\begin{array}{ll}
\vdash_{def} & \mathcal{M}_{\texttt{IntExp}}\ \texttt{(NUM } n\texttt{)} = n\ \wedge \\
& \mathcal{M}_{\texttt{IntExp}}\ \texttt{(IF } b\ i1\ i2\texttt{)} = (\mathcal{M}_{\texttt{BoolExp}}\ b \rightarrow \mathcal{M}_{\texttt{IntExp}}\ i1 \mid \mathcal{M}_{\texttt{IntExp}}\ i2) \\
\vdash_{def} & \mathcal{M}_{\texttt{BoolExp}}\ \texttt{(BOOL } b\texttt{)} = b\ \wedge \\
& \mathcal{M}_{\texttt{BoolExp}}\ \texttt{(EQ } i1\ i2\texttt{)} = (\mathcal{M}_{\texttt{IntExp}}\ i1 = \mathcal{M}_{\texttt{IntExp}}\ i2)
\end{array}$$

Note that $\mathcal{M}_{\texttt{IntExp}}$ and $\mathcal{M}_{\texttt{BoolExp}}$ evaluate to HOL *num*s and *bool*s respectively. The correctness statement can now be stated in HOL as the following goal:

[2] In the course of conducting these proofs they were, in fact, almost completely generalized, and thus for *any* `datatype` declaration it should be possible to deduce the relevant bijection theorems quite quickly.

[3] The use of the symbols $\mathcal{M}_{\texttt{IntExp}}$ and $\mathcal{M}_{\texttt{BoolExp}}$ in the theorems is maintained for clarity, though of course other symbols are used in HOL

goal 1

```
"∀ mliexp mlbexp.
 IS_MLIEXP mliexp ⊃
   EVAL (s, E, mliexp) M_interp_iexp =
      MLNUM_TO_NUM (𝓜_IntExp (MLIEXP_TO_IEXP mliexp)), s ∧
 IS_MLBEXP mlbexp ⊃
   EVAL (s, E, mlbexp) M_interp_bexp =
      MLBOOL_TO_BOOL (𝓜_BoolExp (MLBEXP_TO_BEXP mlbexp)), s"
```

Note the use of the appropriate bijections to convert *mliexp* and *mlbexp* to the correct form needed by the meaning functions, and also to convert the results of the meaning functions back to *Val*s.

The proof for the goal is performed by structural induction over the possible values of *mliexp* and *mlbexp*. The first step is to apply an induction tactic which uses the mutually recursive induction theorem over the two *Val*-subsets. Note that the correctness of both interpretation functions is proved simultaneously. This is absolutely crucial, as the proof could not be performed if the functions were verified separately.

The induction tactic gives us four subgoals, with base cases for MLNum and MLBool and step cases for MLIf and MLEq. For instance, the subgoal for the first base case is

goal 2

```
"IS_MLNUM mlnum ⊃
   EVAL (s, E_I, MLNum mlnum) M_I =
      NUM_TO_MLNUM (𝓜_IntExp (MLIEXP_TO_IEXP (MLNum mliexp)))"
```

We firstly simplify the expected result by proving the following lemma about $\mathcal{M}_{\mathtt{IntExp}}$:

7

```
  ⊢  IS_MLNUM mlnum ⊃
     𝓜_IntExp(MLIEXP_TO_IEXP (MLNum mlnum)) = MLNUM_TO_NUM mlnum
```

After we apply this result the goal can be solved by using the symbolic evaluation tactic `EVAL_TAC`. This will automatically prove that the second match rule of the function `interp_iexp` succeeds, and that the resulting value is *mlnum*.

The remaining subgoals are proved in a similar fashion to this. For the step cases the induction hypotheses allow the correctness of both interpretation functions to be assumed for the recursive calls. The symbolic evaluator is powerful enough to deduce the effect of the `choose` function, despite the fact the it has not been explicitly verified anywhere in our proof. It can also deduce the evaluation theorem for the base value `equal`, based on results about the SML_APPLY function. This completes the proof of correctness for the interpreter.

10.3 Computation Properties

This section moves on from the topic of verification to briefly discuss how the semantics can be used to reason about the language in other ways.

The semantics of Standard ML allow several other interesting definitions of properties of computations to be defined. Firstly, consider termination:

Definition 3 (Termination) *The evaluation of a phrase "ph" in a background consisting of s, E and obj terminates if and only if there exists a result res and state s' such that:*

$$s, E, obj \vdash phrase \Rightarrow res, s'$$

is inferable.

In HOL this is stated as shown below. The second theorem follows from the definition of EVAL.

8

$\vdash_{def}$ TERMINATES $(s,\mathtt{E},obj)$ $phrase$ = $(\exists res\ s'.$ IS$(s,E,obj,phrase,res,s'))$
$\vdash$ TERMINATES (s,E,obj) $phrase$ = $(\exists res\ s'.$ EVAL (s,E,obj) $phrase$ = RESULT $(res,s'))$

We may similarly define what it means for a computation to cause no state changes and no exceptions:

9

$\vdash_{def}$ NO_SIDE_EFFECTS (s,E,obj) $phrase$ =
  $(\forall res\ s'.$ IS$(s,E,obj,phrase,res,s') \supset (s = s'))$
$\vdash_{def}$ NO_EXCEPTIONS (s,E,obj) $phrase$ =
  $(\forall res\ s'\ p.$ IS$(s,E,obj,phrase,res,s') \supset \neg(res =$ Res_Pack $p))$

Note that these definitions are all fundamentally semantic rather than syntactic. For example, it is possible to prove that

```
  let val tmp = !store in
     (store := 6; store := tmp)
  end;
```

causes no side effects.

Given these definitions, it is fairly easy to prove the following results from the correctness theorem for the `append` function:

10

$\vdash$ IS_MLLIST v_1 $\wedge$ IS_MLLIST v_2 $\supset$
  TERMINATES $(s,\ E_{append},$ MLPAIR $v_1\ v_2)$ M_{append}
$\vdash$ IS_MLLIST v_1 $\wedge$ IS_MLLIST v_2 $\supset$
  NO_SIDE_EFFECTS $(s,\ E_{append},$ MLPAIR $v_1\ v_2)$ M_{append}
$\vdash$ IS_MLLIST v_1 $\wedge$ IS_MLLIST v_2 $\supset$
  NO_EXCEPTIONS $(s,\ E_{append},$ MLPAIR $v_1\ v_2)$ M_{append}

Similar results hold for the interpreter that has been have verified.

10.4 The Associativity of append

The last result to be presented in this thesis is a proof of the associativity property of the `append` function, described briefly in Section 2.3.4. This result is not a verification proof, for it asserts the semantic equivalence of two program fragments. Only the case where it may be assumed that each of the arguments to the function evaluate successfully is considered here. The goal is stated in HOL as:

goal 3

```
"IS_MLLIST xval ∧ IS_MLLIST yval ∧ IS_MLLIST zval ∧
 EVAL(s₀, E₀, Obj_NONE) append = Closure_append, s₀ ∧
 EVAL(s₀, E₀, Obj_NONE) X = xval, s₁ ∧
 EVAL(s₁, E₀, Obj_NONE) Y = yval, s₂ ∧
 EVAL(s₂, E₀, Obj_NONE) Z = zval, s₃ ⊃
     IS(s₀, E₀, Obj_NONE, append X (append Y Z), res, s_final) =
     IS(s₀, E₀, Obj_NONE, append (append X Y) Z, res, s_final)
```

Note that in the assumptions we use the predicate EVAL, whereas in the conclusion the notion of inferability is employed. This is merely for clarity, for it better models the statement of the theorem as given in Section 2.3.4. Note also that we have to make an assumption that the variable append does actually evaluate to the correct closure, here denoted by $Closure_{append}$.[4]

The theorem follows from the statement of correctness of the append function. We also need to lift the associativity result for HOL lists:

11

```
# let MLLIST_APPEND_ASSOC = lift_thm APPEND_ASSOC;;
MLLIST_APPEND_ASSOC = ⊢
    IS_MLLIST l1 ∧ IS_MLLIST l2 ∧ IS_MLLIST l3 ⊃
    (MLLIST_APPEND l1 (MLLIST_APPEND l2 l3) =
     MLLIST_APPEND (MLLIST_APPEND l1 l2) l3)
```

This gives us our associativity result for the Standard ML list *Val*-subset. We also prove the following theorem, which is rally just an application level statement of the correctness of the append function:

goal 4

```
SMLAPPEND_APPL = ⊢
   ∀ xval yval s₀ s₁ E₀ s₂ X Y
   IS_MLLIST xval ∧ IS_MLLIST yval ∧
   EVAL(s₀, E₀, Obj_NONE) append = Closure_append, s₀ ∧
   EVAL(s₀, E₀, Obj_NONE) X = xval, s₁ ∧
   EVAL(s₁, E₀, Obj_NONE) Y = yval, s₂ ⊃
      IS(s₀, E₀, Obj_NONE, append X Y, MLLIST_APPEND xval yval, s₂)
```

By specializing this theorem to the appropriate cases and applying the associativity lemma we reduce the goal to the following subgoal:

[4] There is one subtle assumption that initially has to be made but can later be eliminated. The present assumptions give an evaluation for the variable append in the initial state s_0. However the variable also gets evaluated in state s_1, after the evaluation of X. However, it is easy to prove that the evaluation of a variable is always totally independent of the state, unless it is prefixed by the dereferencing operator.

goal 5

"IS(s_0, E_0, Obj_NONE, append X (append Y Z), res, s_{final}) =
 IS(s_0, E_0, Obj_NONE, append (append X Y) Z, res, s_{final})"
   [IS(s_0, E_0, Obj_NONE, $\underline{\texttt{append } X \texttt{ (append } Y\ Z)}$,
         MLLIST_APPEND $xval$ (MLLIST_APPEND $yval$ $zval$), s_2)]
   [IS(s_0, E_0, Obj_NONE, $\underline{\texttt{append (append } X\ Y)\ Z}$,
         MLLIST_APPEND $xval$ (MLLIST_APPEND $yval$ $zval$), s_2)]

The two assumptions give the results of the evaluations for the left and right hand sides of the goal. The result now follows by applying the Determinacy Theorem.

Chapter 11

Conclusions

This chapter summarizes the main achievements of the project and addresses related issues. The suitability of a tool such as HOL for the project is discussed, as are some of the problems that might be faced if the work were used for more complex examples of verification. Lastly, some possibilities for future research are described.

The Project

The aim of this project was to investigate aspects of machine assisted reasoning about the programming language Standard ML. The method used was to construct a HOL theory of the semantics of the language, based on its Formal Definition.

As the project was the continuation of work begun by Matthew Hutchins in 1990, the progress made cannot be measured only in terms of the quantity of results produced. The project was not simply a matter of utilising a set of previously constructed results, but rather each area of Hutchins' theory had to be reworked and overhauled to properly represent the Formal Definition. The major achievements of the project may thus be summarized as follows:

- A complete and logically sound theory of the Standard ML Core dynamic semantics was constructed. This was the main goal of the representation work of Chapters 4, 5 and 6.
- A series of fundamental and important language properties were proved from the semantics. This was done in Chapter 7.
- Special techniques were developed to represent concrete Standard ML types in proofs. This was done in Chapter 8 by characterising the appropriate subsets of the set of all possible Standard ML values.
- Symbolic evaluation was developed further in Chapter 8. It was successfully employed as the fundamental proof techniques for the proofs correctness for the `append` function and the small interpreter in Chapter 10.

In Chapter 4 the main task was to identify and understand the problems in the type theoretical constructs, and then represent the types correctly by using and porting an existing HOL package. The problems faced in defining the semantic types in Chapter 5 were more subtle and complex, and much time was spent constructing proper solutions to problems such as the representation of finite partial functions.

The use of inference trees to define the semantics in Chapter 6 achieved the main goal of ensuring that computations are finite. Techniques learned since have opened other avenues for the possible representation of these rules, but the present representation is sufficient. Defining inferability via an inductive relation may be cleaner.

The importance of the lessons learned in carrying out the first major proofs based on the representation cannot be over emphasized. At the start of the project no techniques for proving theorems from the semantics or even stating correctness were evident, and the uncertain logical foundation of the theory meant that one could not place too much confidence in theorems that might be proved. This situation is now reversed, for the foundations are now sound and some promising techniques are available for verifying programs and proving language properties. Symbolic evaluation seems particularly powerful. It is difficult to imagine other approaches, say ones based on resolution, being as powerful, efficient or flexible.

HOL

Chapters 4 and 5 were mainly concerned with the construction of new HOL types. HOL type theory is not easy to understand, especially for a novice user of the HOL system. However once the basics of primitive recursive functions and induction theorems were grasped, structural type theory became both clear and useful. The application of structural type theory to the representation of Standard ML types in Chapter 9 demonstrates how crucial it is to grasp the theoretical basis of the constructs that are being considered.

When the difficulty of the problems involved is taken into account, HOL has proved to be an excellent tool for the job at hand. Its powerful meta-language means that new proof techniques, and generalizations of existing ones, can be constructed without too much difficulty. Although it takes a long time to learn the HOL system properly, once learnt it becomes surprisingly powerful.

Some possible improvements to the HOL system might be:

- Better support for defining new recursive types and definitions. This would need to include mutually recursive types and more general type domains.
- A debugging system for HOL's ML meta-language.
- An improved set of interface facilities, allowing a wider range of special symbols and visualisations of proof trees. A graphical interface would be ideal for this.

Problems and Future Directions

The possible applications of a properly constructed theory of the Core semantics are wide. With regard to program verification, two small proofs have already been presented in this thesis. However some larger proof examples need to be constructed to discover more of the pragmatic issues involved. The main problem standing in the way of large scale verification proofs may well be the Operational Semantics used in the Formal Definition of Standard ML. In Operational Semantics a function is represented by its code rather than its meaning, and thus it may be hard to prove equivalences and relationships between functions. It would also be difficult to characterise the types of functions in terms of the values that represent them, which may be important if any serious verification is to be performed.

Other possible applications of the theory include proving more about language equivalences, non-side effecting computations and areas such as compiler correctness. In the latter Standard ML could be used as either the source or target language.

In terms of the project itself, several areas were problematic and need more research. Some of potential research topics are:

- The extension of the evaluator to cope with commonly occurring evaluation patterns, particularly those involving boolean types and pairs. This could probably be generalized to any concrete type.
- The relationship between the static and dynamic semantics of the Core.
- The automation of theorem and definition lifting via bijections.
- The automation of defining structured subsets of types.
- The automated definition of recursive types which include functions.
- Completely automating the proof of correctness for simple functions such as those which process lists.
- Investigating the semantics of modules and techniques for proving their correctness.

The solutions to some of these problems may well find application in other HOL proofs and other logic systems.

Lastly, the lessons learnt throughout this project regarding logical consistency, logical representation of types and automation techniques have application in nearly every area of computer science, for they provide excellent examples for understanding what does and does not form a logical foundation for a system.

Appendix A

Types Needed by the Standard Definition

This appendix consists of the following:

- An EBNF description of the syntax of Core language. This definition was used for the definition of the syntactic types in HOL in Chapter 4.
- The definition of the semantic types used in the Formal Definition for the dynamic semantics of the Core. This was used for the definition of the semantic types in HOLin Chapter 5.

Both sections are based on the corresponding portions of the Formal Definition of Standard ML [20]. They have been modified slightly to reflect the fact that input/output, modules, strings and real numbers were not considered in this project.

Core Syntax

This ection gives the syntax of the reduced language considered in this thesis. Keywords are denoted in typewriter font, optional elements are delimited by square brackets.

The basic phrase classes are:

Var	variables
Con	constructors
ExCon	exception constructors
Lab	labels
AtExp	atomic expressions
ExpRow	expression rows
Exp	expressions
Match	matches
MRule	match rules
Dec	declarations
ValBind	value bindings
ExBind	exception bindings

AtPat	atomic patterns
PatRow	pattern rows
Pat	patterns

The language is defined by the following grammar:

var	::=	*identifier*	value variables
con	::=	*identifier*	value constructors
	|	*integer*	integer constants
excon	::=	*identifier*	exception constructors
lab	::=	*identifier*	record labels
atexp	::=	*var*	value variable
	|	*con*	value constructor
	|	*excon*	exception constructor
	|	{ [*exprow*] }	record
	|	**let** *dec* **in** *exp* **end**	local declaration
	|	(*exp*)	
exprow	::=	*lab* = *exp* [, *exprow*]	
exp	::=	*atexp*	atomic
	|	*exp atexp*	application
	|	*exp* **handle** *match*	handle exception
	|	**raise** *exp*	raise exception
	|	**fn** *match*	function
match	::=	*mrule* [| *match*]	
mrule	::=	*pat* => *exp*	
dec	::=	**val** *valbind*	value declaration
	|	**exception** *exbind*	exception declaration
	|	**local** dec_1 **in** dec_2 **end**	local declaration
	|		empty declaration
	|	dec_1 [;] dec_2	sequential declaration
valbind	::=	*pat* = *exp* [**and** *valbind*]	
	|	**rec** *valbind*	
exbind	::=	*excon* [**and** *exbind*]	
	|	*excon* = *excon* [**and** *exbind*]	
atpat	::=	_	wildcard
	|	*var*	variable
	|	*con*	constant
	|	*excon*	exception constant

	$\mid$	$\{$ [*patrow*] $\}$	record
	$\mid$	(*pat*)	
patrow	$::=$	$\ldots$	wildcard
	$\mid$	$lab = pat$ [, *patrow*]	
pat	$::=$	*atpat*	atomic
	$\mid$	*con atpat*	value construction
	$\mid$	*excon atpat*	exception construction
	$\mid$	*var* **as** *pat*	layered

The Semantic Types

All types in the dynamic semantics are built from the syntax classes defined in the previous section together with the simple object classes shown below.

$$
\begin{array}{rcll}
a & \in & \mathit{Addr} & \text{addresses} \\
en & \in & \mathit{ExName} & \text{exception names} \\
b & \in & \mathit{BasVals} & \text{base values} \\
con & \in & \mathit{Con} & \text{constructors} \\
 & & \mathit{FAIL} & \text{failure}
\end{array}
$$

Addr and *ExName* are infinite sets. *BasVal* is the set containing `div`, `mod`, `plus`, `minus`, `times`, `equal`, `notequal`, `less`, `grtr`, `leq` and `geq`. *Con* is identical to the syntactic identifier class. FAIL is a special values used to denote the failing attempt to match a value and a pattern. Thus FAIL is neither a value nor an exception, but simply a semantic object used in the rules to express operationally how matching proceeds.

The compound types are shown below.

$$
\begin{array}{rcrcl}
v & \in & \mathit{Val} & = & \{\texttt{:=}\} \cup \mathit{typeCon} \cup \mathit{BasVal} \cup \mathit{Con} \cup \mathit{Con} \times \mathit{Val} \cup \\
 & & & & \mathit{ExVal} \cup \mathit{Record} \cup \mathit{Addr} \cup \mathit{Closure} \\
r & \in & \mathit{Record} & = & \mathit{Lab} \xrightarrow{\text{fin}} \mathit{Val} \\
e & \in & \mathit{ExVal} & = & \mathit{ExName} \cup (\mathit{ExName} \times \mathit{Val}) \\
p & \in & \mathit{Pack} & = & \mathit{ExVal} \\
C & \in & \mathit{Closure} & = & \mathit{Match} \times \mathit{Env} \times \mathit{VarEnv} \\
mem & \in & \mathit{Mem} & = & \mathit{Addr} \xrightarrow{\text{fin}} \mathit{Val} \\
ens & \in & \mathit{ExNameSet} & = & \mathrm{Fin}(\mathit{ExName}) \\
s & \in & \mathit{State} & = & \mathit{Mem} \times \mathit{ExNameSet} \\
E & \in & \mathit{Env} & = & \mathit{VarEnv} \times \mathit{ExConEnv} \\
VE & \in & \mathit{VarEnv} & = & \mathit{Var} \xrightarrow{\text{fin}} \mathit{Val} \\
EE & \in & \mathit{ExConEnv} & = & \mathit{ExCon} \xrightarrow{\text{fin}} \mathit{ExName}
\end{array}
$$

Appendix B

Proofs

This appendix describes the HOL proofs of the language properties outlined in Chapter 7. The proofs were outlined briefly in that chapter, but are given in more detail here.

B.1 The Backward Rewrite Rules

This section outlines the techniques used in deriving the 43 backward rewrite rules described in Chapter 7.

Proving A Simple Case

As the first example of a HOL proof based on the semantics only a simple backward rewrite rule shall be considered - that for phrases of the form "`Ph_AtExp (AtExp_Var` *var*`)`". The goal that needs to be proven is:

goal 1

$$\begin{array}{l}\texttt{"IS}(s, E, obj, \texttt{Ph\_AtExp (AtExp\_Var } var\texttt{)}, res, s') \;=\\ \quad (\exists val\ VVM\ EE.\\ \qquad (obj = \texttt{Obj\_NONE}) \wedge (res = \texttt{Res\_Val } val) \wedge (s' = s) \wedge\\ \qquad (E = \texttt{Env } VVM\ EE) \wedge (\texttt{APPLY } var\ VVM = \texttt{RESULT } val)\ )\texttt{"}\end{array}$$

We should not be overwhelmed by the conjuncts on the right - they are merely a respecification of the conditions of inferability except without reference to inference trees. Inference trees must, however, be used to prove the goal. The first step is to use `REWRITE_TAC` to expand out the definitions of IS, VALID_INFR, VALID, AtExp_VALID, AtExp_Var_VALID. We also rewrite with the lemma

1

$$\vdash (\texttt{MAP } f\ l = \texttt{[]}) = (l = \texttt{[]})$$

Only the left-hand side of the equality is affected, where the IS predicate is replaced by the side conditions that need to hold for the sentence to be inferable. The new goal is:

goal 2

```
"(∃ch.
   (∃val VVM EE.
     (ch = []) ∧
     (obj = Obj_NONE) ∧ (res = Res_Val val) ∧ (s' = s) ∧
     (E = Env VVM EE) ∧ (APPLY var VVM = RESULT val)) ∧
   EVERY VALID_INFR ch) =
(∃val VVM EE.
   (obj = Obj_NONE) ∧ (res = Res_Val val) ∧ (s' = s) ∧
   (E = Env VVM EE) ∧ (APPLY var VVM = RESULT val) )"
```

Here ch represents the sub-inferences to the node of an inference tree which has the sentence as its root. Of course, for a base case such as this, ch will have to be empty. The goal is now obviously provable, but we shall show the rest of the proof to illustrate some important proof techniques which allow us to prove such goals very quickly.

Variable Elimination and Choosing Existential Witnesses

The goal we now have is an equality between two existentially quantified boolean formulae, and the best course open to us is to prove the "if" and "only if" cases of the equivalence separately. The HOL tactic EQ_TAC breaks $P = Q$ into the two subgoals $P \Rightarrow Q$ and $Q \Rightarrow P$. The sub-goals may then be "stripped" using STRIP_TAC, which moves terms on the left of an implication into the assumptions. STRIP_TAC also skolemizes existentially quantified terms as they are moved into the assumptions. We put EQ_TAC and STRIP_TAC together with the tactical THEN, which applies two tactics in succession, and the tactical REPEAT which applies a tactic repeatedly until an exception occurs. Thus the HOL tactic EQ_TAC THEN REPEAT STRIP_TAC splits a goal and strips both implications as much as possible. Applying this to the case at hand, two subgoals are produced, the first of which is:

goal 3

```
"∃val' VVM' EE'.
  (obj = Obj_NONE) ∧ (res = Res_Val val') ∧ (s' = s) ∧
  (E = Env VVM' EE') ∧ (APPLY var VVM' = RESULT val')"
    [ "ch = []" ]
    [ "obj = Obj_NONE" ]
    [ "res = Res_Val val" ]
    [ "s' = s" ]
    [ "E = Env VVM EE" ]
    [ "APPLY var VVM = RESULT val" ]
    [ "EVERY VALID_INFR ch" ]
```

Two things are interesting about this goal. Firstly, 5 of the assumptions are simple assertions of the form "*var* = *term*", e.g. "obj = Obj_NONE". It would be helpful if we could simply "eliminate" the variables from the goal, for we know from the assumptions what value they have. Thus wherever we have *obj* we should be able to simply substitute Obj_NONE. The need to perform variable elimination will occur over and over again in our proofs, and it is helpful to write a special tactic which performs it accurately and efficiently. The tactic `SMART_ELIMINATE_TAC` does this and is described in Appendix C. Using `REPEAT SMART_ELIMINATE_TAC` reduces the above goal to:

goal 4

```
"∃val' VVM' EE'.
  (Obj_NONE = Obj_NONE) ∧ (Res_Val val = Res_Val val') ∧ (s = s) ∧
  (Env VVM EE = Env VVM' EE') ∧ (APPLY var VVM' = RESULT val')"
    [ "APPLY var VVM = RESULT val" ]
    [ "EVERY VALID_INFR []" ]
```

The second thing of interest is just how "obvious" the values needed for *val'*, *VVM'* and *EE'* are. The HOL built-in tactic `EXISTS_TAC` *tm* removes existential quantifiers from goals, with the term *tm* being substituted as the existential witness. For instance `EXISTS_TAC "val"` would be the next obvious step in solving the above goal, leaving a subgoal where `"val'"` has been substituted by `"val"`. Manually specifying these witnesses is a time-consuming and error-prone task. What we would really like is to be able to tell the system to "use the witnesses that are obviously available from the goal we are trying to prove." The tactic `SMART_EXISTS_TAC` was designed and implemented as part of this project to do just that. The tactic is a highly improved and optimized version of the `SML_EXISTS_RULE` developed by Hutchins for his work.

We now use `REPEAT SMART_EXISTS_TAC` to reduce the proof to the subgoal:

goal 5

```
"(Obj_NONE = Obj_NONE) ∧ (Res_Val val = Res_Val val) ∧ (s = s) ∧
 (Env VVM EE = Env VVM EE) ∧ (APPLY var VVM = RESULT val)"
    [ "APPLY var VVM = RESULT val" ]
```

which is of course trivial - each of the conjuncts in the goal are either syntactic equalities or assumptions. The goal is solved by applying `ASM_REWRITE_TAC` which rewrites the goal using the assumptions. Thus the entire first subgoal may be proven with the tactic:

```
  REPEAT SMART_ELIMINATE_TAC
  THEN REPEAT SMART_EXISTS_TAC
  THEN ASM_REWRITE_TAC []
```

The other subgoal in the proof is proven by essentially the same tactic. Note the simplicity of the proof and how it contains no instance-specific constructs. This is generally the sign of a well-constructed HOL proof, for the same tactics may be applied to similar goals.

A More Complicated Case

The above proof has been for a very simple case, but demonstrates that we can prove that inferability may be stated without direct appeal to inference trees. We shall now consider the proof for a more complicated case, when sub-inferences contribute to the nodes of the inference tree. For the Exp_Raise case our goal is:

goal 6

```
"IS(s,E,obj,Ph_Exp(Exp_Raise exp),res,s') =
 (∃exval.
   (obj = Obj_NONE) ∧ (res = Res_Pack(Pack exval)) ∧
   IS(s,E,Obj_NONE,Ph_Exp exp,Res_Val(Val_ExVal exval),s')) ∨
 (∃p.
   (obj = Obj_NONE) ∧ (res = Res_Pack p) ∧
   IS(s,E,Obj_NONE,Ph_Exp exp,Res_Pack p,s'))"
```

The theorem is stating the two possible cases for the evaluation of an expression

`raise` *exp*

Either the evaluation of *exp* succeeds, in which case an exception is raised with the value of *exp*, or the evaluation causes an exception, and this exception is passed on. The two cases are represented by the disjuncts on the right.

The first two steps of the proof are similar to the simple case. We rewrite with the definitions of IS, VALID_INFR, VALID, Exp_VALID, Exp_Raise_VALID and some helpful lemmas, use `EQ_TAC` to split the boolean equality and then use `REPEAT STRIP_TAC` to simplify. Now `STRIP_TAC` transforms the goal $(\Gamma, (P \vee Q) \Rightarrow R)$[1] into two subgoals $(\Gamma \cup P, R)$ and $(\Gamma \cup Q, R)$. Each side of the rewritten goal has a disjunct, and thus stripping our goal produces *four* subgoals.

Constructing the Sub-Inferences

We will consider the third subgoal, which corresponds to the case where we are proving an inference tree exists when we now the first disjunct on the right to be true. After variable elimination the goal has the following form:

goal 7

```
"∃ch. ( ...  ∨  ... ) ∧ EVERY VALID_INFR ch"
   [ "VALID_INFR(Node(s,E,Obj_NONE,Ph_Exp exp,Res_Val(Val_ExVal exval),s')ch)" ]
```

We are now required to find a value for `ch`, which represents the children to the inference node of the inference tree we are constructing. The children are supplied by the predicate in the assumptions, which says exactly which children we know to be inferred. However, a tactic such as `SMART_EXISTS_TAC` is not clever enough to realise that

ch = `"[s,E,Obj_NONE,Ph_Exp exp,Res_Val(Val_ExVal exval),s']"`

is the appropriate choice. Thus, we must supply the children explicitly.

At this point we should remember that a similar proof is going to be needed for at least 2 of the 4 subgoals in our proof, and indeed for half of the subgoals in each of the 43 theorems we are trying to prove - about 100 subcases in all! Manually specifying the children would be impossible, and so we look for a way to create a proof technique which generalises this step for all of the cases.

In essence what we want to do is "construct the list of children from the assumptions". Once we know what we want, it is easy to write a tactic which does just that:

[1] Γ represents the assumptions to the goal.

```
let construct_ch_tac (asms,gl) =
   let inferred_ch = mapfilter dest_VALID_INFR asms in
   let chldrn = mk_list(inferred_ch, type_of (hd inferred_chldrn)) in
   let hyps = map ASSUME asms in
   let thm = PROVE ("EVERY VALID_INFR ^chldrn", REWRITE_TAC (EVERY_DEF.hyps)) in
         (EXISTS_TAC chldrn THEN REWRITE_TAC [thm]) (asms,gl);;
```

Here `asms` is a list of terms representing the assumptions to the goal, and `gl` is the goal itself. We bind `inferred_ch` to the list of sentences in the assumptions which are used in VALID_INFR predicates - in our example there is only one. We then bind `chldrn` to a HOL list constructed from these sentences using `mk_list`. The assumptions as theorems are contained in `hyps`, and `thm` is a theorem proved in a backwards fashion which asserts that "EVERY VALID_INFR" is true for the list of children we have just constructed. Finally, we select `chldrn` as our existential witness for the quantified variable *ch*, and then rewrite with the theorem we have just proved.

Applying `construct_ch_tac` gives us a goal of the form:

goal 8

```
"(∃exval' ch'.
   (Res_Pack(Pack exval) = Res_Pack(Pack exval')) ∧
   ([Node(s,E,Obj_NONE,Ph_Exp exp,Res_Val(Val_ExVal exval),s')ch] =
    [Node(s,E,Obj_NONE,Ph_Exp exp,Res_Val(Val_ExVal exval'),s')ch'])) ∨
 (∃p ch'.
   (Res_Pack(Pack exval) = Res_Pack p) ∧
   ([Node(s,E,Obj_NONE,Ph_Exp exp,Res_Val(Val_ExVal exval),s')ch] =
    [Node(s,E,Obj_NONE,Ph_Exp exp,Res_Pack p,s')ch']))"
  [ "VALID_INFR(Node(s,E,Obj_NONE,Ph_Exp exp,Res_Val(Val_ExVal exval),s')ch)" ]
```

The `construct_ch_tac` is able to construct the correct children list for all of the subgoals we want to prove.

Choosing Disjuncts using Sentence Signatures

We shall use this last goal to demonstrate another specialised proof technique that we shall use quite often in the remainder of this chapter. The goal is disjunctive, and it is obvious that the first disjunct is provable while the second is not. Thus the correct tactic to apply is `DISJn_TAC 1` which chooses the first disjunct as a subgoal. However, as with constructing the children, we do not want to have to specify the particular disjunct for each particular proof we write. Instead, we would like to automate the process of choosing the disjunct. Note that at this point in the proof we always have enough information to make an appropriate choice, in that one and only one of the disjuncts will be provable.

Automating the choice is best done by modeling the human reasoning process involved in deciding which disjunct to use. In the general case, the disjunct to choose depends entirely upon the inferable sentences we have available to us in the assumptions. We look at the inferable sentences available, the nodes in the disjuncts, and perform a match - the disjunct which seems to match is the one we choose. We model this reasoning process in the following way. Firstly, we define a function which computes a *signature* for a sentence. The signature must carry enough information to characterize any of the sentences that occur in the semantics, in other words it is what the sentence "looks like". Typically the type of phrase and the type of result is sufficient, but to allow quite general signatures we a list of constructors. Applying this to our problem at hand, we compute the signatures in the assumptions. This models the process of looking at the assumptions to see what types of sentences we have inferred:

2

```
# sentence_signature;;
sentence_signature : term -> term list

# sentence_signature "(s,E,Obj_NONE,Ph_Exp exp,Res_Val (Val_ExVal exval),s')";;
["Ph_Exp";"Res_Val";"Val_ExVal"] : term list
```

We now compute the signatures in the different disjuncts of the goal. This is complicated by the fact we are only interested in sentences on the right of the list equalities, but we can tell the difference by the free variables in the sentences. For our current goal:

3

```
# search_for_signatures (top_goal());;
[ ["Ph_Exp";"Res_Val";"Val_ExVal"]; ["Ph_Exp";"Res_Pack"] ]: term list list
```

We now compare the signatures, matching sets. Our assumptions' signatures match with the first set of signatures from the goal, and thus we know to choose the first disjunct. An outline of the code for the automation of this process is shown below:

```
let choose_disj_tac (asms,gl) =
    let asms_sigs = map search_for_signatures asms
    and disj_sigs = map search_for_signatures (disjuncts gl) in
       (DISJn_TAC (match_signatures disj_sigs asms_sigs)) (asms,gl);;
```

Applying `choose_disj_tac` reduces the goal to one disjunct which is easily proved using `SMART_EXISTS_TAC` and `ASM_REWRITE_TAC`.

It would be pointless implementing this technique if we had only one sub-goal, but it is in fact powerful enough to choose the correct disjunct for *all* the different subgoals of *all* 43 axioms we wish to prove. The technique is used in several other proofs also.

Automating Rewriting

The one step we have not yet generalised is the initial rewrite using the definition of inference trees. We need to automatically select the correct definition theorems to use with the rewriting tactic, for it would be extremely inefficient to rewrite with all 46 definitions from the previous chapter. We develop a special tactic `SML_IS_REWRITE_TAC` to search the goal, decide on an appropriate set of rewrites and do the rewriting. We shall not give the details of the tactic here. For the second proof example we considered, the definitions used would be `IS_DEF`, `VALID_INFR_DEF`, `VALID_DEF`, `Exp_VALID_DEF` and `Exp_Raise_VALID_DEF`.

The Final Tactic

We can now write our general tactic to prove each of the backward rules. An outline of the tactic is shown below. The real tactic is slightly more complicated as it includes a handful of lemmas regarding lists and trees.

```
let prove_bwdrule_tac =
   SML_IS_REWRITE_TAC THEN EQ_TAC
   THENL [
      REPEAT STRIP_TAC THEN REPEAT SMART_ELIMINATE_TAC
      THEN choose_disj_tac
   ;  REPEAT STRIP_TAC THEN REPEAT SMART_ELIMINATE_TAC
      THEN construct_ch_tac
      THEN choose_disj_tac
   ]
   THEN REPEAT (SMART_EXISTS_TAC THEN ASM_REWRITE_TAC []);;
```

This tactic is both simple and robust because we have modeled the logic behind the proofs, rather than relying on adhoc peculiarities of the HOL system. Each proof is essentially similar due to the uniform way in which the semantics have been modeled using disjunctive and existential statements. The tactic has survived several substantial changes to the semantics without requiring major modification. In addition, we now have the techniques of signature searching, variable elimination and existential witness guessing available for future use.

B.2 The Pattern Matching Theorem

This section describes in detail the proof of the Pattern Matching Theorem as outlined in Chapter 7. The goal is stated as follows:

goal 9

"IS(s, E, rec, pat, res, s') $\supset$ (s = s')"

The result is proven by structural induction over the syntax of patterns. Firstly we must generate a simplified version of the induction theorem for the entire Standard ML syntax:

```
let Pat_PatRow_AtPat_INDUCT = prove_mut_rec_subset_induction_thm
                  [":Pat";":PatRow";":AtPat"] Syntax_INDUCT;;
```

The theorem produced is shown in Figure B.1. It proves three predicates simultaneously over the pattern grammar, one each for *Pat*s, *AtPat*s and *PatRow*s. We must instantiate the induction theorem with the predicate we want to prove for each particular type. The predicate we want to prove for *Pat*s is simply our goal. The predicate for *AtPat*s is a simple modification of this:

goal 10

"IS(s, E, val, $atpat$, res, s') $\supset$ (s = s')"

The predicate for *PatRow*s is similar. We instantiate the induction theorem with these predicates, match the goal with the conclusion of the induction theorem using the HOL tactic `MATCH_MP_TAC`, and then strip the goal. This gives us 14 sub-goals corresponding to the 4 base cases and 10 step cases in the induction proof.

The first of these subgoals is the base case for wildcard rows:

$\vdash \forall P\_PatRow\ P\_AtPat\ P\_Pat.$
$\quad P\_PatRow$ `WildRow` $\wedge$
$\quad (\forall P.\ P\_Pat\ P \supset (\forall L.\ P\_PatRow($`PatRow` $L\ P))) \wedge$
$\quad (\forall P0\ P1.$
$\quad\quad$ `P_Pat` $P0 \wedge P\_PatRow\ P1 \supset (\forall L.\ P\_PatRow($`PatRow_And` $L\ P0\ P1))) \wedge$
$\quad P\_AtPat$ `Wildcard` $\wedge$
$\quad (\forall V.\ P\_AtPat($`AtPat_Var V`$)) \wedge$
$\quad (\forall C.\ P\_AtPat($`AtPat_Con` $C)) \wedge$
$\quad (\forall E.\ P\_AtPat($`AtPat_ExCon` $E)) \wedge$
$\quad P\_AtPat$ `AtPat_Record_Empty` $\wedge$
$\quad (\forall P.\ P\_PatRow\ P \supset P\_AtPat($`AtPat_Record` $P)) \wedge$
$\quad (\forall P.\ P\_Pat\ P \supset P\_AtPat($`AtPat_Paren` $P)) \wedge$
$\quad (\forall A.\ P\_AtPat\ A \supset P\_Pat($`Pat_AtPat` $A)) \wedge$
$\quad (\forall A.\ P\_AtPat\ A \supset (\forall C.\ P\_Pat($`Pat_Con` $C\ A))) \wedge$
$\quad (\forall A.\ P\_AtPat\ A \supset (\forall E.\ P\_Pat($`Pat_ExCon` $E\ A))) \wedge$
$\quad (\forall P.\ P\_Pat\ P \supset (\forall V.\ P\_Pat($`Pat_Layered` $V\ P))) \supset$
$\quad (\forall x.\ P\_PatRow\ x) \wedge (\forall x.\ P\_AtPat\ x) \wedge (\forall x.\ P\_Pat\ x)$

Figure B.1: The Induction Theorem for Patterns

goal 11

"IS(s, E, Obj_Rec rec, WildRow, res, s') $\supset$ ($s = s'$)"

We have already proved a backward rewrite rule for wildcard rows:

4

$\vdash$ IS(s,E,obj,WildRow,res,s') $=$
$\quad$ ($\exists rec$. (obj = Obj_Rec rec) $\wedge$ (res = Res_VarEnv ZIP) $\wedge$ (s' = s))

Rewriting with this rule gives us the goal:

goal 12

"($\exists rec'$.
$\quad$ (Obj_Rec rec = Obj_Rec rec') $\wedge$
$\quad$ (res = Res_VarEnv ZIP) $\wedge$
$\quad$ (s' = s)) $\supset$
(s = s')"

We strip, and the result is proved immediately for one of the conjuncts in the assumptions is the goal we want to prove. We now consider a step case:

goal 13

"IS(s,E,Obj_Rec rec,Ph_PatRow (PatRow L P),res,s') $\supset$ (s = s')"
$\quad$ ["$\forall E\ s\ val\ res\ s'$. IS($s$,$E$,Obj_Val val,Ph_Pat P,res,s') $\supset$ (s = s')"]

The assumption is our induction hypotheses, where we have assumed that our goal holds for the sub-pattern P. Rewriting with the appropriate backward rewrite rules gives:

goal 14

```
"(∃VE val.
   (Obj_Rec rec = Obj_Rec(EXT(L,val)ZIP)) ∧ (res = Res_VarEnv VE) ∧
   IS(s,E,Obj_Val val,Ph_Pat P,Res_VarEnv VE,s')) ∨
 (∃val.
   (Obj_Rec rec = Obj_Rec(EXT(L,val)ZIP)) ∧ (res = Res_FAIL) ∧
   IS(s,E,Obj_Val val,Ph_Pat P,Res_FAIL,s')) ⊃
 (s = s')"
    [ "∀E s val res s'. IS(s,E,Obj_Val val,Ph_Pat P,res,s') ⊃  (s = s')"]
```

We now again strip the goal, and eliminate variables using `SMART_ELIMINATE_TAC`. We get two sub goals because of the disjunct in the assumption, the first of which is:

goal 15

```
"(s = s')"
    [ "∀E s val res s'. IS(s,E,Obj_Val val,Ph_Pat P,res,s') ⊃  (s = s')"]
    [ "IS(s,E,Obj_Val val,Ph_Pat P,Res_VarEnv VE,s')" ]
```

The result now follows immediately by applying the HOL resolution tactic `RES_TAC`.

It is obvious that the proofs for each of the induction subcases will follow a similar pattern, and so we can now write a general tactic which will prove each. The basic structure of the tactic is:

```
let PROVE_INDUCTION_CASE_TAC =
   SML_IS_BWD_REWRITE_TAC
   THEN REPEAT STRIP_TAC
   THEN REPEAT SMART_ELIMINATE_TAC
   THEN RES_TAC
   THEN RES_TAC
```

The tactic `SML_IS_BWD_REWRITE_TAC` is similar to the `SML_IS_REWRITE_TAC` described in the previous section, in that it automates the selection of an appropriate set of backward rewrite rules to use in rewriting a goal.

Finally, we have the proof of the entire theorem, shown below. The proof takes approximately 1 minute on a Sun-4:

5

```
# let PATTERN_MATCHING = PROVE(
#    "IS(s, Obj_Env_Rec E rec, Ph_Pat pat, res, s') ==> (s = s')",
#    MULTI_MATCH_INDUCT_TAC Pat_PatRow_AtPat_INDUCT
#    THEN REPEAT CONJ_TAC
#    THEN PROVE_INDUCTION_CASE_TAC );;
PATTERN_MATCHING = |- "IS(<s>,<E>,<rec>,<pat>,<res>,<s'>) ==> (<s> = <s'>)"
```

`MULTI_MATCH_INDUCT_TAC` is a small tactic developed to streamline the application of the relevant induction theorem to the goal.

B.3 The Forward Inference Rules

This section outlines the proof techniques employed in proving the forward inference rules of Chapter 7. These were originally proved by Matthew Hutchins in [12], and the reader may wish to refer to that work also.

We shall show the proof for only one rule - a forward rule for Exp_Raise phrases. We aim to prove the goal:

goal 16

```
"IS(s,E,Obj_NONE,Ph_Exp exp,Res_Val(Val_ExVal exval),s') ⊃
 IS(s,E,Obj_NONE,Ph_Exp(Exp_Raise exp),Res_Pack(Pack exval),s')"
```

The theorem states that if the expression *exp* evaluates to the extra exception information *exval* then the expression "`raise` *exp*" evaluates to an exception packet with *exval* attached.

To prove the goal, we first strip the implication and then rewrite the right-hand side of the implication using backward rewrite rules. We use the `SML_IS_BWD_REWRITE_TAC` developed in the previous section for this purpose. The new goal is:

goal 17

```
"(∃exval'.
   (Res_Pack(Pack exval) = Res_Pack(Pack exval')) ∧
   IS(s,E,Obj_NONE,Ph_Exp exp,Res_Val(Val_ExVal exval'),s')) ∨
 (∃p.
   (Res_Pack(Pack exval) = Res_Pack p) ∧
   IS(s,E,Obj_NONE,Ph_Exp exp,Res_Pack p,s'))"
    [ "IS(s,E,Obj_NONE,Ph_Exp exp,Res_Val(Val_ExVal exval),s')" ]
```

As in section 7.1 we have a disjunctive goal where one disjunct is obviously provable and the other is not. We may apply a version of `choose_disj_tac`, again matching sentence signatures between the assumptions and the goal. In this situation the tactic will choose the first disjunct:

goal 18

```
"∃exval'.
   (Res_Pack(Pack exval) = Res_Pack(Pack exval')) ∧
   IS(s,E,Obj_NONE,Ph_Exp exp,Res_Val(Val_ExVal exval'),s')"
    [ "IS(s,E,Obj_NONE,Ph_Exp exp,Res_Val(Val_ExVal exval),s')" ]
```

This result is now solved by applying `SMART_EXISTS_TAC` and the rewriting. An outline of the tactic which proves all 86 rules follows:

```
let prove_forward_rule_tac =
      REPEAT STRIP_TAC
      THEN SML_IS_BWD_REWRITE_TAC
      THEN choose_disj_tac
      THEN REPEAT SMART_EXISTS_TAC
      THEN ASM_REWRITE_TAC [];;
```

B.4 The Determinacy Theorem

This section outlines the proof of the determinacy theorem as described in Chapter 7.

We state our goal as follows:

goal 19

```
"IS(s,env,obj,ph,res,s') ∧ IS(s,env,obj,ph,res',s'') ⊃
     (res = res') ∧ (s' = s'')"
```

We prove the following lemma first:

goal 20

```
"∀n.
   ISDB n (s,env,obj,ph,res,s') ∧
   ISDB n (s,env,obj,ph,res',s'') ⊃
         (res,s' = res',s'')"
```

To prove the lemma we can induct on n, which gives us two subgoals. With some simple arithmetic the case $n = 0$ quickly is solved easily, for it can be proven that $\forall tree.$ Depth $tree \geq 1$. Thus we are left with the step case of the induction:

goal 21

```
"ISDB (SUC n) (s,env,obj,ph,res,s') ∧
 ISDB (SUC n) (s,env,obj,ph,res',s'') ⊃  (res,s' = res',s'')"
   [ ∀s env obj ph res s'.
       ISDB n (s,env,obj,ph,res,s') ∧
       ISDB n (s,env,obj,ph,res',s'') ⊃  (res,s' = res',s'')" ]
```

We can now work with individual cases of the kind of phrase ph we are dealing with, giving us 43 subgoals - one for each kind of phrase in the system. The first of these is:

goal 22

```
"ISDB (SUC n) (s,env,obj,Ph_AtExp (AtExp atexp),res,s') ∧
 ISDB (SUC n) (s,env,obj,Ph_AtExp (AtExp atexp),res',s'') ⊃
    (res = res') ∧ (s' = s'')"
    [ "∀ph s''' env obj res res' s' s''.
        ISDB n (s''',env,obj,ph,res,s') ∧
        ISDB n (s''',env,obj,ph,res',s'') ⊃  (res = res') ∧ (s' = s'')" ]
```

The first method that comes to mind to prove each of the subgoals of the induction is to rewrite ISDB in terms of its definition, and then rewrite using the definition of VALID_INFR, VALID and so on. This will work, but a better approach is available, based on a new set of backward rewrite rules. These rules can be seen as a series of lemmas for this proof, but in fact have wider application.

Backward Rewrite Rules and Computational Induction

Our first set of backward inference rules rewrote an inferable sentence in terms of its sub-inferences. For the purposes of this proof they are inadequate, for they fail to capture the information that the depths of the inference trees for the sub-inferences are less than the depth of the inference tree for the entire case. However we may restate the theorems using ISDB and adding an extra condition that the depths do in fact decrease: Our two example rewrite rules will become:

6

```
⊢ ISDB n (s,E,obj,Ph_AtExp(AtExp_Var var),res,s') =
   (∃val VVM EE.
     1 = n ∧
     (obj = Obj_NONE) ∧ (res = Res_Val val) ∧ (s' = s) ∧
     (E = Env VVM EE) ∧ (APPLY var VVM = RESULT val))

⊢ ISDB n (s,E,obj,Ph_Exp(Exp_Raise exp),res,s') =
   (∃exval.
     (obj = Obj_NONE) ∧ (res = Res_Pack(Pack exval)) ∧
     ISDB (PRE n) (s,E,Obj_NONE,Ph_Exp exp,Res_Val(Val_ExVal exval),s')) ∨
   (∃p.
     (obj = Obj_NONE) ∧ (res = Res_Pack p) ∧
     ISDB (PRE n) (s,E,Obj_NONE,Ph_Exp exp,Res_Pack p,s'))
```

The theorems are identical to the backward rewrite rules, except that some extra depth information has been included. The first theorem states that a sentence with phrase AtExp_Var *var* will be inferable with depth n if and only of $n \leq 1$ and the normal side conditions for inferability hold. The second theorem states a similar result, except it demands that its sub-inference have depth one less than n, written as PRE n.

The importance of this new set of results is undoubtable:

- Induction over the depth of a computation is a common and powerful technique for proving language properties. As such, it is likely that these results are going to be needed more than once.

- It is advantageous to eliminate any need to talk about inference trees in our proofs, and instead talk only about inferability. The backward inference rules allowed us to do this for all cases except those which rely in some way on the depth of inference. The new rules allow us to do it for all other cases.

- Together the new rules characterize the semantics completely, for they guarantee that the inference process is finite.

There are 43 depth backward rewrite rules in total, and as with the backward rewrite rules we may organize their proofs such that one tactic proves all. In the proofs the techniques of child construction, variable elimination, existential witness guessing, disjunct choosing and automatic rewriting get used extensively again.

Proving the Sub-Goals

Once the depth rewrite rules have been proved, the proof of each of the subgoals for the Determinacy Lemma follows fairly quickly. We shall show two subcases[2]. The first subcase

[2]Unfortunately the important subcase for function application is too large for us to show here.

was shown above, and we rewrite using the appropriate depth rewrite rule and the lemma $\vdash$PRE(SUCn)=n , then strip and eliminate variables:

goal 23

"Res_Val val, s = Res_Val val, s"

The goal is now proved by rewriting.

The second subcase is again for phrases of the form Ph_Exp (Exp_Raise exp):

goal 24

"ISDB(SUC n)(s, env, obj,Ph_Exp(Exp_Raise E'),res, s') $\wedge$
 ISDB(SUC n)(s, env, obj,Ph_Exp(Exp_Raise E'),res', s'') $\supset$
 (res, s' = res', s'')"
    ["$\forall ph\ s\ env\ obj\ res\ res'\ s'\ s''$.
        ISDB $n(s, env, obj, ph, res, s')$ $\wedge$
        ISDB $n(s, env, obj, ph, res', s'')$ $\supset$
        (res, s' = res', s'')"]

Rewriting and stripping gives 4 subgoals. Of these, two represent contradictions where the paths of the two computations are not identical, and two represent coinciding paths. In either case, each subgoal is proved by repeatedly resolving with the induction hypothesis, eliminating variables, choosing existential witnesses and then rewriting.

The proof of all 43 subgoals of the Determinacy Lemma can be streamlined through the following tactic:

```
let mono_stepcase_tac =
   SML_ISDB_REWRITE_TAC [PRE]
   THEN REPEAT STRIP_TAC2
   THEN REPEAT (RES_TAC THEN SMART_ELIMINATE_TAC
                THEN SMART_EXISTS_TAC THEN ASM_REWRITE_TAC [])
```

Some extra lemmas are need to prove some pathological cases.

Deriving the Theorem from the Lemma

The derivation of the theorem from the lemma was discussed in Chapter 7.

Appendix C

Using HOL

The appendix presents two HOL tactics which were developed in the course of this project, as well as a set of helpful strategies and hints for using the HOL system. The tactics act as case studies for how general purpose proof strategies can be built in HOL once the need for them is identified. The motivations for the tactics were presented briefly in Appendix B.

Both tactics, and the *fpf* theory constructed in Chapter 5, have been submitted to the HOL contributors directory for the next release of the HOL system.

SMART_EXISTS_TAC

This tactic is motivated by the desire to be able to automatically prove goals of the form "$\exists x_0$ ... x_n. ...". The usual approach to solving such a goal is to introduce appropriate values for $x_0 \ldots x_n$ into the proof (they are often already present), and then use `EXISTS_TAC` to supply existential witnesses to the goal. However, in most HOL goals appropriate choice for one or more of the variables is "obvious" nine times out of ten. A good example of this was shown in Appendix B.

The strategy used by the tactic is to firstly search the goal for equality terms, keeping note of how the free and bound variables change. For example, for the goal

"$\exists x\ y.$ P $x \wedge$ (y = 3)"

the tactic will choose the value `3` for y. This demonstrates one of the most powerful features of the tactic - the ability to find values for nested existential values. The tactic will reduce the above goal to:

"$\exists x.$ P $x \wedge$ (3 = 3)"

If no appropriate values can be found in the goal, it will look through the assumptions for a value. For instance:

"$\exists x\ y.$ P $x \wedge$ Q y"
   [Q (1 + 2)]

will result in the choice of `1 + 2` for y.

The tactic is quite intelligent in the sense that it will nearly always choose the correct term for the goal, provided sufficient information is present in either the assumptions or the goal. The implementation of the tactic is interesting, for the search strategy employed must ensure that only sensible choices are made. The HOL meta-language is ideal for implementing a search such as this.

SMART_ELIMINATE_TAC

One of the most frustrating types of goals to attempt to solve using the primitive facilities of the HOL system are ones containing equalities in the assumptions. For instance, consider the following step in proving a transitivity-like porperty for $<$:

```
"a < d"
    [ "b = c" ]
    [ "b = a" ]
    [ "c < d" ]
```

The tactic `ASM_REWRITE_TAC` is not useful in this situation, because the equalities are asserted the wrong way around. In addition, even when the equalities are useful, rewriting with assumptions may be extremely slow and may also perform extra unwanted rewrites. One solution is to do some sort of filtered rewriting, but this is generally difficult to construct and often hard to generalise.

In proofs such as the above, what we would really like to do is simply eliminate the variable b from the proof altogether. Mentally this is what we do anyway. Furthermore we would like to remove the assumptions about b from the proof, because they only serve to clutter the proof at later stages. Note that this contravenes the "natural" style of proof, where assumptions are only ever enriched. In a situation such as this this seems entirely appropriate. Of course HOL ensures that the process is still logically valid.

The tactic `SMART_ELIMINATE_TAC` will reduce the above goal to

```
"a < d"
    [ "a = c" ]
    [ "c < d" ]
```

by eliminating b, and then

```
"a < d"
    [ "a < d" ]
```

by eliminating c.

The tactic is able to cope with quite complex substitution patterns efficiently. It is also able to eliminate terms rather than variables when this is appropriate. In particular, terms may be eliminated if the free variables they contain are not used in any other context in the proof other than additional occurences of the term. This leads to tactics `SMART_VAR_ELIMINATE_TAC`, `SMART_TERM_ELIMINATE_TAC` which prevent and allow the elimination of terms. The elimination tactics end up being so useful that one's automatic reaction to seeing an equality in the assumptions of a goal is to apply the appropriate one.

Helpful Hints and Strategies

The following is a partial list of some of the lessons learnt about the HOL system in the course of this project. The list is designed to aid future users of the system to make better use of its features:

Theorem Naming It is absolutely crucial that a consistent set of rules be used to name theorems, definitions and axioms. When many hundreds of theorems are present, it is important to be able to access the correct theorems quickly and efficiently.

Autoloading HOL provides a very useful autoloading facility, which allows theorems and definitions to be extracted from theory files and bound to ML identifier names automatically. This saves the entire theory being loaded, and the tiresome process of specifying exactly which theorems are wanted is avoided. A refinement was made to this facility as part of this project. A program was written to automatically generate a `".use"` file from a theory. The file contains all the necessary autoloading commands.

Learning Forward Proof Techniques Although it is complex, it is well worth the time to learn the forward proof techniques available to the user in the HOL system. Backward proof will always remain the most common proof style used in HOL, but the need to specify goals exactly and then construct proofs can be frustrating, especially for simple theorems that "obviously" follow from existing theorems, say by a simple rewrite.

Using Libraries Whenever a library or package is used, it is worth investigating fully its features. Often libraries will provide the proof of many difficult theorems, as well as supplying tactics and proof rules related to the theory. This are both useful and serve as good examples of how to automate proofs in HOL.

Appendix D

The Finite Partial Functions Theory

In this appendix the *fpf* theory constructed in Chapter 5 is shown. A proof for each of the theorems listed was constructed in HOL. The purpose in including this theory is to act as a reference for potential users of the theory, and also as an example of the amount of work that goes into constructing HOL theories properly.

The Theory *fpf*

Types –

fpf

Constants –

SUCCEEDS	TRANSFORM_P	EXT_REP	REP_fpf
FAILS	DOM	IS_fpf_REP	LIST_TO_FPF
FAILURE	EXT	RANGE	UNEXT
ABS_fpf	PORTION	RESULT	UNEXT_P
ZIP	EXTBY	RESULTOF	EVERYF
APPLY	EXTBY_P	ZIP_REP	
TRANSFORM	DOM_P	RANGE_P	

Definitions –

FAILURE_DEF $\vdash_{def}$ FAILURE = INR one

RESULT_DEF $\vdash_{def}$ RESULT = INL

SUCCEEDS_DEF $\vdash_{def}$ SUCCEEDS = ISL

```
FAILS_DEF        ⊢def FAILS = ISR

RESULTOF_DEF     ⊢def RESULTOF = OUTL

ZIP_REP_DEF      ⊢def ZIP_REP = (λ x. FAILURE)

EXT_REP_DEF      ⊢def ∀x y map.
                      EXT_REP(x,y)map = (λ x′. ((x = x′) →  RESULT y | map x′))

IS_fpf_REP       ⊢def ∀fpf.
                      IS_fpf_REP fpf =
                      (∀P.
                        P ZIP_REP ∧ (∀fpf′ x y. P fpf′ ⊃  P(EXT_REP(x,y)fpf′)) ⊃
                        P fpf)

ZIP_DEF          ⊢def ZIP = ABS_fpf(λ x. FAILURE)

EXT_DEF          ⊢def ∀x y map.
                      EXT(x,y)map =
                      ABS_fpf(λ x′. ((x = x′) →  RESULT y | REP_fpf map x′))

APPLY_DEF        ⊢def ∀x map. APPLY x map = REP_fpf map x

PORTION_DEF      ⊢def ∀f f′.
                      PORTION f f′ =
                      (∀x y. (APPLY x f = RESULT y) ⊃  (APPLY x f′ = RESULT y))

EXTBY_P          ⊢def ∀f map map′.
                      EXTBY_P f map map′ =
                      (∀x.
                        APPLY x f =
                        (SUCCEEDS(APPLY x map) →  APPLY x map | APPLY x map′))

EXTBY_DEF        ⊢def ∀map map′. EXTBY map map′ = (εf. EXTBY_P f map map′)

TRANSFORM_P      ⊢def ∀map′ fn map.
                      TRANSFORM_P map′ fn map =
                      (∀x.
                        APPLY x map′ =
                        (SUCCEEDS(APPLY x map) →
                         RESULT(fn(RESULTOF(APPLY x map))) |
                         FAILURE))

TRANSFORM_DEF    ⊢def ∀fn map. TRANSFORM fn map = (εmap′. TRANSFORM_P map′ fn map)

DOM_P            ⊢def ∀dom map.
                      DOM_P dom map = (∀x. x IN dom = SUCCEEDS(APPLY x map))

DOM_DEF          ⊢def ∀map. DOM map = (εdom′. DOM_P dom′ map)

RANGE_P          ⊢def ∀ran map.
                      RANGE_P ran map =
                      (∀y. y IN ran = (∃x. APPLY x map = RESULT y))
```

RANGE_DEF	$\vdash_{def}$ ∀map. RANGE map = (εran′. RANGE_P ran′ map)
LIST_TO_FPF_DEF	$\vdash_{def}$ (LIST_TO_FPF[] = ZIP) ∧ (∀pr t. LIST_TO_FPF(CONS pr t) = EXT pr(LIST_TO_FPF t))
UNEXT_P	$\vdash_{def}$ ∀x fpf1 fpf2. UNEXT_P x fpf1 fpf2 = (∀x′. ((x = x′) → (APPLY x′ fpf2 = FAILURE) | (APPLY x′ fpf1 = APPLY x′ fpf2)))
UNEXT_DEF	$\vdash_{def}$ ∀x fpf. UNEXT x fpf = (εfpf′. UNEXT_P x fpf fpf′)
EVERYF_DEF	$\vdash_{def}$ ∀P map. EVERYF P map = (∀x. x IN (DOM map) ⊃ P(x,RESULTOF(APPLY x map)))

Theorems –

INL_11	⊢ ∀x x′. (INL x = INL x′) = (x = x′)
INR_11	⊢ ∀x x′. (INR x = INR x′) = (x = x′)
RESULT_11	⊢ ∀x x′. (RESULT x = RESULT x′) = (x = x′)
SUCCEEDS_RESULT	⊢ ∀x. SUCCEEDS(RESULT x)
NOT_SUCCEEDS_FAILURE	⊢ ¬SUCCEEDS FAILURE
RESULTOF_RESULT	⊢ ∀x. RESULTOF(RESULT x) = x
FAILS_FAILURE	⊢ FAILS FAILURE
NOT_FAILS_RESULT	⊢ ∀x. ¬FAILS(RESULT x)
fpf_INDUCT	⊢ ∀P. P ZIP ∧ (∀fpf. P fpf ⊃ (∀x y. P(EXT(x,y)fpf))) ⊃ (∀fpf. P fpf)
APPLY_ZIP	⊢ ∀x. APPLY x ZIP = FAILURE
NOT_SUCCEEDS_APPLY_ZIP	⊢ ∀x. SUCCEEDS(APPLY x ZIP) = F
APPLY_EXT	⊢ ∀x y v f. APPLY x(EXT(y,v)f) = ((y = x) → RESULT v | APPLY x f)
APPLY	⊢ (∀x. APPLY x ZIP = FAILURE) ∧ (∀x y v f. APPLY x(EXT(y,v)f) = ((y = x) → RESULT v | APPLY x f))

```
EQ_IMP_APPLY_EXT        ⊢ ∀x y v f. (y = x) ⊃  (APPLY x(EXT(y,v)f) = RESULT v)

NE_IMP_APPLY_EXT        ⊢ ∀x y v f.
                            ¬(y = x) ⊃  (APPLY x(EXT(y,v)f) = APPLY x f)

fpf_EQ                  ⊢ ∀f1 f2. (f1 = f2) = (∀x. APPLY x f1 = APPLY x f2)

NOT_INL_EQ_INR          ⊢ ∀x y. ¬(INL x = INR y)

NOT_RESULT_EQ_FAILURE   ⊢ ¬(RESULT x = FAILURE)

NOT_FAILURE_EQ_RESULT   ⊢ ¬(FAILURE = RESULT x)

NOT_EXT_ZIP             ⊢ ∀x y f. (EXT(x,y)f = ZIP) = F

NOT_ZIP_EXT             ⊢ ∀x y f. (ZIP = EXT(x,y)f) = F

EXT_EXT                 ⊢ ∀x u v f. EXT(x,u)(EXT(x,v)f) = EXT(x,u)f

EXT_COMM                ⊢ ∀x y u v f.
                            ¬(x = y) ⊃
                            (EXT(x,u)(EXT(y,v)f) = EXT(y,v)(EXT(x,u)f))

fpf_SING_EQ             ⊢ ∀x y x′ y′.
                            (EXT(x,y)ZIP = EXT(x′,y′)ZIP) ⊃
                            (x = x′) ∧ (y = y′)

fpf_PAIR_EQ             ⊢ ∀x x′ y y′ y″ y‴.
                            ¬(x = x′) ⊃
                            (EXT(x,y)(EXT(x′,y″)ZIP) =
                             EXT(x,y′)(EXT(x′,y‴)ZIP)) ⊃
                            (y = y′) ∧ (y″ = y‴)

fpf_SND_ABSORPTION      ⊢ ∀fpf d y.
                            SUCCEEDS(APPLY d fpf) = (∃_unique y. EXT(d,y)fpf = fpf)

PORTION_SELF            ⊢ ∀f. PORTION f f

PORTION_ZIP             ⊢ ∀f. PORTION f ZIP = (f = ZIP)

APPLY_EXTBY             ⊢ ∀f1 f2 x.
                            APPLY x(EXTBY f1 f2) =
                            (SUCCEEDS(APPLY x f1) →  APPLY x f1 | APPLY x f2)

EXTBY                   ⊢ (∀f. EXTBY ZIP f = f) ∧
                          (∀x y f1 f2.
                            EXTBY f1(EXT(x,y)f2) =
                            (SUCCEEDS(APPLY x f1) →
                             EXTBY f1 f2 |
                             EXT(x,y)(EXTBY f1 f2))) ∧
                          (∀x y f1 f2.
                            EXTBY(EXT(x,y)f1)f2 = EXT(x,y)(EXTBY f1 f2))
```

```
APPLY_TRANSFORM        ⊢ ∀fn map x.
                           APPLY x(TRANSFORM fn map) =
                           (SUCCEEDS(APPLY x map) →
                            RESULT(fn(RESULTOF(APPLY x map))) |
                            FAILURE)

TRANSFORM              ⊢ (∀fn. TRANSFORM fn ZIP = ZIP) ∧
                          (∀fn fpf x y.
                            TRANSFORM fn(EXT(x,y)fpf) =
                            EXT(x,fn y)(TRANSFORM fn fpf))

IN_DOM                 ⊢ ∀map x. x IN (DOM map) = SUCCEEDS(APPLY x map)

DOM                    ⊢ (DOM ZIP = EMPTY) ∧
                          (∀fpf x y. DOM(EXT(x,y)fpf) = x INSERT (DOM fpf))

EMPTY_DOM              ⊢ ∀f. (DOM f = EMPTY) = (f = ZIP)

IN_DOM_IMP_APPLY       ⊢ ∀fpf x.
                           x IN (DOM fpf) ⊃  (∃y. APPLY x fpf = RESULT y)

IN_DOM_EQ_APPLY        ⊢ ∀fpf x. x IN (DOM fpf) = (∃y. APPLY x fpf = RESULT y)

NOT_IN_DOM_IMP_APPLY   ⊢ ∀fpf x. ¬x IN (DOM fpf) ⊃  (APPLY x fpf = FAILURE)

EXT_ABSORPTION         ⊢ ∀f x.
                           ∃f'. ∀y.
                             (EXT(x,y)f = EXT(x,y)f') ∧
                             (∀x'. APPLY x' f' = ((x' = x) →  FAILURE | APPLY x' f)) ∧
                             (DOM f' = (DOM f) DELETE x)

DOM_FINITE             ⊢ ∀f. FINITE(DOM f)

CARD_DOM_INSERT        ⊢ ∀x f.
                           CARD(x INSERT (DOM f)) =
                           (x IN (DOM f) →  CARD(DOM f) | SUC(CARD(DOM f)))

CARD_DOM_EQ_0          ⊢ ∀x f. (CARD(DOM f) = 0) = (f = ZIP)

CARD_EQ_SUC            ⊢ ∀i s.
                           FINITE s ⊃
                           (CARD s = SUC i) ⊃
                           (∃s' x.
                             ¬x IN s' ∧ (CARD s' = i) ∧ (s = x INSERT s'))

CARD_DOM_SUC           ⊢ ∀f i.
                           (CARD(DOM f) = SUC i) ⊃
                           (∃f' x y.
                             ¬SUCCEEDS(APPLY x f') ∧
                             (CARD(DOM f') = i) ∧
                             (f = EXT(x,y)f'))
```

```
STRONG_INDUCTION_lemma  ⊢ ∀P.
                              P 0 ∧ (∀j. (∀i. i = j ⊃  P i) ⊃  P(SUC j)) ⊃
                        (∀n i. i = n ⊃  P i)

STRONG_INDUCTION        ⊢ ∀P.
                              P 0 ∧ (∀j. (∀i. i = j ⊃  P i) ⊃  P(SUC j)) ⊃
                        (∀n. P n)

fpf_CARD_INDUCT         ⊢ ∀P.
                            (∀f. (f = ZIP) ⊃  P f) ∧
                            (∀j.
                              (∀i. i = j ⊃  (∀f. (CARD(DOM f)) = i ⊃  P f)) ⊃
                         (∀f. (CARD(DOM f)) = (SUC j) ⊃  P f)) ⊃
                        (∀f. P f)

fpf_INDUCT_2            ⊢ ∀P.
                            P ZIP ∧
                            (∀fpf x y.
                              P fpf ⊃
                              ¬SUCCEEDS(APPLY x fpf) ⊃
                              P(EXT(x,y)fpf)) ⊃
                            (∀fpf. P fpf)

IN_RANGE                ⊢ ∀fpf y.
                            y IN (RANGE fpf) = (∃x. APPLY x fpf = RESULT y)

RANGE_ZIP               ⊢ RANGE ZIP = EMPTY

RANGE_EXT               ⊢ ∀fpf x y.
                            ¬x IN (DOM fpf) ⊃
                            (RANGE(EXT(x,y)fpf) = y INSERT (RANGE fpf))

EMPTY_RANGE             ⊢ ∀f. (RANGE f = EMPTY) = (f = ZIP)

SUCCEEDS_OR_FAILS       ⊢ ∀x fpf. SUCCEEDS(APPLY x fpf) ∨ FAILS(APPLY x fpf)

SUCCEEDS_EQ_NOT_FAILS   ⊢ ∀x fpf. SUCCEEDS(APPLY x fpf) = ¬FAILS(APPLY x fpf)

APPLY_UNEXT             ⊢ ∀x fpf x'.
                            ((x = x') →
                             (APPLY x'(UNEXT x fpf) = FAILURE) |
                             (APPLY x' fpf = APPLY x'(UNEXT x fpf)))

UNEXT_ZIP               ⊢ ∀x. UNEXT x ZIP = ZIP

APPLY_UNEXT_SAME        ⊢ ∀x fpf. APPLY x(UNEXT x fpf) = FAILURE

APPLY_UNEXT_DIFF        ⊢ ∀x fpf x'.
                            ¬(x = x') ⊃  (APPLY x' fpf = APPLY x'(UNEXT x fpf))

UNEXT_EXT               ⊢ ∀fpf x x' y.
                            UNEXT x(EXT(x',y)fpf) =
                            ((x = x') →  UNEXT x fpf | EXT(x',y)(UNEXT x fpf))

EVERYF_ZIP              ⊢ ∀P. EVERYF P ZIP = T

EVERYF_EXT              ⊢ ∀P fpf d r.
                            EVERYF P(EXT(d,r)fpf) =
                            P(d,r) ∧ EVERYF P(UNEXT d fpf)

EVERYF                  ⊢ (∀P. EVERYF P ZIP = T) ∧
                          (∀P fpf d r.
                            EVERYF P(EXT(d,r)fpf) =
                            P(d,r) ∧ EVERYF P(UNEXT d fpf))

EVERYF_TRUTH            ⊢ ∀f. EVERYF(λ p. T)f = T

EVERYF_PTRUTH           ⊢ ∀f. EVERYF(λ (x,y). T)f = T
```

Appendix E

Extracts from the HOL Theories of the Formal Definition

This appendix lists some small portions of the theories developed in Chapter 4, 5 and 6. These are included as references to these chapters. Unfortunately space does not allow us to show the entire theories.

Appendix F

The HOL Theories of a Standard ML Type

This appendix shows one of the theories that results from the construction of a Standard ML type as a *Val*-subset in HOL. The case example is for Standard ML lists. The construction of the theory was described briefly in Chapter 9.

Bibliography

[1] Mark Aargaard and Miriam Lesser. A formally verified system for logic synthesis. *Masters Thesis*, April 1992. School of Electrical Engineering, Cornell University.

[2] Richard Bird and Philip Wadler. *Introduction to Functional Programming.* Prentice Hall, 1988.

[3] University Of Calgary. *HOL90 - System Notes*, 1992.

[4] A. Cant and M.A. Ozols. A verification environment for ML. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, pages 151–155, June 1992.

[5] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[6] R.L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, 1986.

[7] Michael J. C. Gordon. *The Denotational Description of Programming Languages: An Introduction.* Springer-Verlag, 1979.

[8] Michael J. C. Gordon. Mechanizing programming languages in Higher Order Logic. *HOL System Documentation*, 1988.

[9] Michael J. C. Gordon et al. *The HOL System Description*, December 1989.

[10] Brian Graham. Dealing with the Choice operator in HOL88. *HOL System Documentation*, 1990.

[11] Robert W. Harper. Standard ML input/output. *Polymorphism*, 2(2), 1985.

[12] Matthew Hutchins. Machine assisted reasoning about Standard ML using HOL. *Honours Thesis*, November 1990. Australian National University.

[13] David MacQueen. Modules for Standard ML. *Polymorphism*, 2(2), 1985.

[14] Robin Milner. Lecture notes on language semantics. Computer Science 3, University of Edinburgh.

[15] Robin Milner. How ML evolved. *Polymorphism*, 1(1), 1983.

[16] Robin Milner. The Standard ML Core language. *Polymorphism*, 2(2), 1985.

[17] Robin Milner and Mads Tofte. *Commentary on Standard ML*. MIT Press, 1990.

[18] Larry Paulson. *ML For the working programmer*. MIT Press, 1992.

[19] Chris Reade. *Elements of Functional Programming*. Addison Wesley, 1989.

[20] Robin Milner Robert Harper and Mads Tofte. *The Definition Of Standard ML, Version 2*. MIT Press, 1990.

[21] Claudio V. Russo. Automating mutually recursive type definitions in HOL. *Honours Thesis*, June 1992. University Of Edinburgh.