AR-006-933

DSTO
AUSTRALIA

ELECTRONICS RESEARCH LABORATORY

Information Technology Division

RESEARCH REPORT
ERL-0600-RR

PROGRAM VERIFICATION USING HIGHER ORDER LOGIC

by

A. Cant

SUMMARY

This paper describes a number of experiments in program verification carried out within two automated proof assistants, namely the HOL (Higher Order Logic) system and Isabelle. Various approaches to programming language semantics are described. Theories and tactics for proving the correctness of programs written in small functional and imperative languages are then constructed within HOL and Isabelle.

JAN 92

APPROVED FOR PUBLIC RELEASE

POSTAL ADDRESS: Director, Electronics Research Laboratory, PO Box 1500, Salisbury, South Australia, 5108. ERL-0600-RR

UNCLASSIFIED

CONTENTS

LIST OF TABLES

Page No.

Chapter 1 INTRODUCTION

The study of computing machines has always ranged from the fundamental mathematical approach, such as computability and decidability, to more practical engineering concerns, such as the construction of integrated circuits. The discipline of computer programming sits (at times uneasily) between science and engineering. In the past, programming tended to be regarded as something of a black art. The early programming languages (such as FORTRAN) made it almost impossible to write well-structured and easy-to-understand programs. If a program happened to work without crashing, one was relieved.

Modern computer science is becoming more and more a mathematical discipline, with the realisation that languages can be carefully designed to enable well-structured programs to be written, and that mathematics is an appropriate arena for reasoning about computer programs. Today computer science and software development are increasingly making use of *formal methods*, namely specification languages, automated theorem provers, verification tools and a number of other systems based on formal mathematical techniques.

This paper focuses on the issue of program verification: proving that programs written in a given language conform to their specifications, ie are *correct*.

1.1 Why Program Verification?

Large software systems are now widespread, being used in finance, medicine, defence, transport, power generation and in other application areas. In many of these systems, one failure could be disastrous, leading to loss of life, damage to the environment, breaches of national security etc. We refer to these systems generally as *critical* systems, because failures can lead to critical hazards — those with an unacceptable risk. In *safety-critical* software, for example, attention is focused on the possibility of serious injury or even loss of life if the software fails.

Software, by its nature, can be extremely complex. In many cases it is not possible to have an overall mental grasp of what a piece of software does. Although modern software engineering stresses the use of suitable programming languages and techniques, errors will inevitably occur in the software. Testing is the time-honoured way of removing these errors. Sometimes, it does not matter if software has errors — for example, the software used to write this paper has numerous bugs, but none of these prevent useful results from being obtained.

But in a critical system, just one simple error somewhere in the code might have catastrophic consequences. In the USA, the FAA and NASA have established a requirement of less than 10^{-10} safety-critical failures per hour throughout a ten hour flight [1]. For hardware components, it is possible to achieve such low failure rates by using highly reliable components and redundancy in the design. For software, the situation is different. Current software testing techniques can reduce this figure to perhaps 10^{-4}per hour. Testing can make significant contributions to the assurance of safety; however, the reliability levels obtained by testing fall far short of the levels required for critical applications. In any case, the complexity and logical nature of software means that a probabilistic approach is of doubtful validity.

Therefore, in such cases, testing is not adequate (Dijkstra's comment that testing can show the presence of bugs, but never their absence, is well-known). We need a way of verifying that the software is correct before it is used.

At this stage, we introduce some terminology. Software can be termed *correct* only with respect to a detailed *formal specification* which describes exactly what the program is required to do. The objective of *program verification* is to prove mathematically that every execution of the program will satisfy the given specifications.

The term *partial correctness* is used to describe a program which either does not terminate, or else is correct with respect to its specifications; the term *total correctness* is used to describe a program which is correct, and also terminates.

In practice, it is also desirable that the formal specifications themselves (which can be complex and subject to logical and other errors) be proved correct with respect to simpler, more abstract specification.

In this case, we speak of *design verification*. Again, this specification may itself need to be verified against an even simpler and yet more abstract specification, until we eventually reach the formal top-level specification. The specifications reflect different levels of detail in the design. Ultimately, we need to check whether the specifications capture the original informal user requirements. This step is called *validation*. The issues of design verification and validation will not be considered further in this paper.

Henceforth, we shall use the term *verification* as a shorthand for program verification.

1.2 Aim of the Paper

The verification of a program is a difficult task — much harder than programming itself. It is an essentially mathematical task. System specifications are naturally expressed in a mathematical notation, and the process of verification uses techniques of mathematical proof.

It follows that, if we are to say mathematically that a program behaves correctly with respect to its specifications, then we must have a clear mathematical description of *what* the program does. In other words, the *semantics* of the language needs to be clearly formulated so that the meaning of constructs in the language is well-understood.

We need to construct a mathematical model of the system by formulating a theory based on certain axioms, and proving theorems from these axioms. The mathematician may be content with — and convinced by — paper and pencil proofs. However, mathematical models of critical systems need to be subjected to a greater level of rigour. In such cases an automated reasoning tool can help us formulate the theory, manage the proofs of key results and avoid logical errors, leading to increased assurance of correctness.

It is the interaction of the language semantics with the issue of program verification which this paper concentrates on. Our aim is to construct a program verification system for a language L, which:

1. is clearly based on semantics of L;
2. is easy to modify;
3. can evaluate programs and expressions (i.e. act as an interpreter if required);
4. can carry out non-trivial reasoning about programs;
5. can prove some non-trivial programs correct;
6. compares well with other systems;
7. is efficient, easy to use;
8. is applicable to concurrent programming languages; and
9. can help assess language design and semantics.

The plan of this paper is as follows. We shall first describe in Chapter 2 various approaches to programming language semantics. In Chapter 3, because of its importance for the theorem proving tools we use, the language ML will be described. In Chapter 4 we give an overview of the HOL system, followed by some examples of verification for a small language. Chapters 6 and 7 deal with the theorem prover Isabelle and with examples of verification. Finally, in Chapter 8 we make some comments on the theorem provers used, and make some suggestions for further work.

Chapter 2 PROGRAMMING LANGUAGE SEMANTICS

2.1 Introduction

There are three characteristics of programming languages [2]:

1. **syntax:** the structure and appearance of legal phrases in the language;
2. **semantics:** the assignment of meanings to the phrases; and
3. **pragmatics:** the areas of application for the language, methods of implementation etc.

Of these three aspects, the first two are directly relevant for program verification. The specification of language syntax is by now well-understood, and Backus Naur form (BNF) is routinely used to describe syntax. The semantics of the language is more difficult to describe, and there is no single method in widespread use. Language semantics is of central importance for program verification. We must have a reliable and usable definition of the semantics in order to talk about program correctness.

2.2 Methods for semantics specification

There are at least four general methods for describing the semantics of a language, which reflect the rather different ways in which a program can be viewed, ranging from highly abstract mathematical constructs to changes in the physical state of hardware devices. The study of programming language semantics is a fascinating and rapidly growing area of modern computer science.

The method of *denotational semantics* maps each phrase in a language (and, in particular, a program) directly to its meaning (called its *denotation*), which is a mathematically defined object, often a number or a function. Thus the phrase is regarded as having a meaning even if no interpreter or compiler for the language exists.

The method of *operational semantics*, sometimes called *natural semantics*, gives the meaning of a program in terms of an interpreter for the language. It defines a program in the language in terms of a sequence of interpreter configurations. The semantics is given as a number of inference rules which describe under what conditions a language construct will evaluate to a particular value.

In the method of *axiomatic semantics*, the meaning of a program is not explicitly given. Rather, logical properties about language constructs are defined, and a number of axioms and inference rules describe under what conditions an assertion about a construct will follow. The most familiar example of the axiomatic method is Hoare logic, which captures assertions about the partial correctness of imperative languages.

Finally, an approach which is essentially equivalent to the denotational description is the method of *algebraic semantics* [3]. In this case, one studies many-sorted algebras and functions from these algebras to possible spaces of meanings. A familiar analogy in mathematics is that of group theory: groups may be studied as abstract objects, and the various representations of the group in (say) finite-dimensional vector spaces can also be studied. Although a most interesting area, under investigation by a number of theoretical computer scientists (including a strong group in France), it has not yet been applied to real programming languages. We shall not be examining this approach further in this paper.

Although these various methods all attack the problem of assigning meaning to programs, they are really complementary rather than competing methods. Whichever approach one chooses depends on particular aspects of the language one is studying. Broadly speaking, we have the following hierarchy [2]:

axiomatic semantics → denotational semantics → operational semantics
language design → language development → language implementation

2.3 Denotational Semantics

We shall now give an overview of denotational semantics using a couple of examples. The subject is a large one, and we do not have the space to give more than a brief description here. There are a number of textbooks on the subject. Gordon [4] gives an elementary introduction which, however, glosses over the deeper mathematical ideas; Stoy's book [5] is an excellent account of the subject. The book by Schmidt [2] is perhaps the best and most up-to-date, while the review article by Mosses [6] is excellent, notably for his attempt to standardise the notation. These last two accounts contain numerous useful references.

Denotational semantics establishes a canonical definition for a language, and thereby documents the design of that language. It also establishes a standard for implementations of the language. Also, and importantly for us, a denotational semantics definition provides a basis for reasoning about the correctness of programs, either directly, or else indirectly, by means of derived proof rules.

Denotational semantics, by its nature, is built on a number of mathematical concepts and some special notation. These will be introduced as needed.

2.3.1 History

The systematic study of denotational semantics was begun in 1964 [7] by Christopher Strachey in the Programming Research Group, University of Oxford. Strachey was using the type-free lambda-calculus to assign meaning to programming constructs. In the words of Mosses [6]:

"By 1969, Dana Scott [the logician] had become interested in Strachey's ideas. In an exciting collaboration with Strachey, Scott first convinced Strachey to give up the type-free lambda calculus; then he discovered that it did have a model after all. Soon after that, Scott established the theory of semantic domains, providing adequate foundations for the semantic descriptions that Strachey had been writing."

Scott's contribution was to put the entire formalism of denotational semantics on a sound mathematical footing, using the theory of domains to make legitimate the definitions of while loops, recursive functions and recursively defined domains. Their remarkable collaboration continued until Strachey's untimely death in 1975.

2.3.2 Syntax

We need to understand programming language syntax in order to formalise semantics correctly. Concrete syntax regards a language as a set of strings over some alphabet, while abstract syntax regards a language as a set of derivation trees. Denotational semantics is solely concerned with abstract syntax. It is the job of a parser to describe how to get from the concrete to the abstract syntax (the reverse transformation is called unparsing, or pretty-printing). The concrete syntax is obtainable from the abstract syntax in a well-defined way, and is not important, save to make programs more readable (we shall see later on that this i: a very real concern in modelling semantics in HOL and Isabelle).

It will be useful to introduce as a working example a small imperative language (essentially the language discussed by Schmidt) which we shall call IMP1. Its abstract syntax is given below:

ABSTRACT SYNTAX OF IMP1

p : Program
c : Command
e : Expression
b : Booleanexpr
i : Identifier
n : Number

$p ::= c$
$c ::= c_1 ; c_2 \mid \text{if} \ \text{then} \ c_1 \ \text{else} \ c_2 \mid i := e \mid \text{diverge} \mid \text{skip} \mid \text{while} \ b \ \text{do} \ c$
$e ::= e_1 + e_2 \mid i \mid n$
$b ::= e_1 = e_2 \mid \neg b \mid \text{true} \mid \text{false}$

In the above, for example, the notation "e : Expression" means that Expression is a syntactic domain, and that e is the non-terminal representing an arbitrary element of that domain (this is a set-theoretic view) or, if we prefer, we can think of "e : Expression" as stating that e is an arbitrary element of type Expression (a type-theoretic view). The latter view is a natural one when we use a typed logic such as HOL to formulate semantics, as we shall see. (However, we must be careful to distinguish types in the logic from types in the language being studied, if any).

The syntax is formulated in terms of a *context-free grammar* (CFG). The BNF rule for a typical expression e says that an expression can either be the sum of two other expressions, an identifier, or a number. In the grammar, objects such as Number and Identifier have no BNF rules associated with them; they are tokens.

It is well-known that certain features of a language (such as typing information and declarations of variables before they are used), cannot be handled by a context-free grammar; these are *context-sensitive* features. In denotational semantics, it turns out to be more convenient to regard such features not as part of syntax, but as part of the semantics, called *static semantics* because it only depends on the program text, and not on how the program might behave at run-time. In this paper, we shall only be concerned with run-time behaviour (i.e.*dynamic* semantics).

2.3.3 Semantic Algebras

Denotational semantics is built on domain theory, due to Scott. The fundamental concept in domain theory is that of a *semantic algebra*, which consists of an underlying set (called a *semantic domain*), along with a number of operations on the domain. A basic example of a semantic algebra is the set Nat of natural numbers, along with the operations plus, minus and times. Certain special domains are necessary to allow the modelling of such things as while loops and recursive data types.

From primitive domains, a number of compound domains can be obtained by means of domain constructors. These are (if A and B are domains)

1. Product domain $A \times B$
2. Sum domain $A + B$ (also known as the disjoint union)
3. Function domain $A \rightarrow B$
4. Lifted domain: $A_{\perp}$

The first three constructions are well-known. The fourth one adds a special element called $\perp$ (read "bottom") which denotes **nontermination** or **undefined**. It is needed to model nonterminating computations and partial functions. A function f: $A_\perp \rightarrow B_\perp$ is called *strict* if $f(\perp) = \perp$. If λx.a is some function on A, then we use the notation $\underline{\lambda}$x.a to denote the extension to a strict function on $A_\perp$.

Recursively defined domains, such as

$$\text{Value} = \text{Number} + (\text{Value} \rightarrow \text{Value})$$

need special handling. The central role of domain theory is to make sense of such equations. It turns out that the above equation has a solution provided that the functions in the domain A $\rightarrow$ B are restricted to be those which are continuous with respect to a certain topology (called the Scott topology) on A and B. Imposing restrictions onto a set in order to make rigorous concepts which already seem intuitively clear is a well-known technique in mathematics (examples are the theory of distributions, or generalised functions, and the theory of quantum mechanical operators) However, the technique is perhaps less familiar in computer science.

We now return to our example IMP1. Imperative languages use a special data structure called a *store*, which exists independently of any program in the language. Certain constructs can access and update the store. In IMP1, we can think of the store as computer memory.

The semantic algebras for IMP1 are the basic domains in which the meaning of our language constructs will be defined. We shall model the store as the domain of functions from identifiers to values. In the table below, $b \Rightarrow x \,\Box\, y$ is a shorthand for "if b then x else y"

SEMANTIC ALGEBRAS FOR IMP1

Truth values
Domain : $\mathrm{Tr} = \{\mathrm{true}, \mathrm{false}\}$
Operations :
not : $\mathrm{Tr} \rightarrow \mathrm{Tr}$

Identifiers
Domain : Ide

Natural Numbers
Domain : Nat
Operations :
plus : $\mathrm{Nat} \times \mathrm{Nat} \rightarrow \mathrm{Nat}$
equals : $\mathrm{Nat} \times \mathrm{Nat} \rightarrow \mathrm{Nat}$

Store
Domain : $\mathrm{Store} = \mathrm{Ide} \rightarrow \mathrm{Nat}$
Operations :
newstore : Store
newstore $= \lambda i.0$

access : $\mathrm{Ide} \rightarrow \mathrm{Store} \rightarrow \mathrm{Nat}$
access $i\ s = s(i)$

update : $\mathrm{Ide} \rightarrow \mathrm{Nat} \rightarrow \mathrm{Store} \rightarrow \mathrm{Store}$
update $i\ n\ s = (\lambda j.j = i \Rightarrow n \,\square\, s(j))$

2.3.4 Semantics

The final step is to assign a meaning to each phrase in the language by giving a valuation (or meaning) function from the phrase in terms of elements and operations of the semantic domains. The definition is an inductive one, guided by the abstract syntax of the language. The key property is that the meaning of a phrase is defined solely in terms of the meaning of its proper subphrases (this is called *compositionality*). For example, the meaning of the phrase "if b then e_1 else e_2" will depend on the meanings of the boolean expression b, and the other expressions e_1 and e_2. There are a number of primitive phrases, such as assignment, which do not depend on any other phrases.

In the case of IMP1, it is natural to picture a command as an operation taking a given store to a new store. Thus its meaning will be given by a function of the following type:

$$C : \mathrm{Command} \rightarrow \mathrm{Store}_{\perp} \rightarrow \mathrm{Store}_{\perp}$$

The Store domain is lifted because the action on the store may not terminate. It is natural to take C to be a strict function on $\mathrm{Store}_{\perp}$, i.e. we cannot recover from an nonrecoverable situation.

The semantics for IMP1 is given below — omitting, for the moment, the `while` construct, which will be discussed later.

SEMANTICS FOR IMP1

$P : \text{Program} \rightarrow \text{Nat} \rightarrow \text{Nat}_{\perp}$

$P[c] = \lambda n.\text{let } s = (\text{update input } n \text{ newstore}) \text{ in let } s' = C[c]s \text{ in } (\text{access output } s')$

$C : \text{Command} \rightarrow \text{Store} \rightarrow \text{Store}$

$C[c_1; c_2] = \underline{\lambda} s.C[c_1](C[c_2]s)$

$C[\text{if } b \text{ then } c_1 \text{ else } c_2] = \underline{\lambda} s.B[b]s \Rightarrow C[c_1]s \square C[c_2]s$

$C[i := e] = \underline{\lambda} s.\text{update } i\ E[e]s\ s$

$C[\text{diverge}] = \underline{\lambda} s.\perp$

$E : \text{Expression} \rightarrow \text{Store} \rightarrow \text{Nat}$

$E[e_1 + e_2] = \lambda s.E[e_1]s \text{ plus } E[e_2]s$

$E[i] = \lambda s.\text{access } i\ s$

$E[n] = \lambda s.n$

$B : \text{Booleanexpr} \rightarrow \text{Store} \rightarrow \text{Tr}$

$B[e_1 = e_2] = \lambda s.E[e_1]s \text{ equals } E[e_2]s$

$B[\neg b] = \lambda s.\neg(B[b]s)$

$B[\text{true}] = \text{true}$

$B[\text{false}] = \text{false}$

The above valuation functions are what we would expect. The semantic functions have as their arguments phrases of the language, enclosed in square brackets for readability. Purely for convenience, the equation for P says that the meaning of a program is obtained by taking an input number, associating it with the special identifier 'input', evaluating the body of the program, and then extracting the answer from the identifier 'output'. Note that an expression does not have side-effects in its evaluation; it may need to consult the store to be evaluated, but will never change it.

As a simple example of working with denotational semantics we have:

$$P[\text{output} := 1;\ \text{if input} = 0 \text{ then diverge};\ \text{output} := 3] = \lambda n.n \text{ equals } 0 \Rightarrow \perp \square\ 3$$

The proof is straightforward (it is treated in detail in [2]). For this program, if the input is 0, the answer is undefined; in all other cases the answer is 3.

Once we have captured the denotational semantics of a language, there is an immediate notion of *equivalence* of expressions and commands (and hence programs). We define:

$$e_1 \approx e_2 \Leftrightarrow E[e_1] = E[e_2]$$
$$c_1 \approx c_2 \Leftrightarrow C[c_1] = C[c_2]$$

This is an important notion: in proving the correctness of a program p we may wish first to apply a transformation T in order to simplify it. If p ≈ T(p), then the correctness (or otherwise) of p will be preserved by the transformation. Here are some examples of equivalences (where c, c_1, c_2 are arbitrary commands and b is an arbitrary boolean expression):

$$e_1 + e_2 \approx e_2 + e_1$$
$$(\ c;\ \text{skip}) \approx (\text{skip};\ c) \approx c$$
$$\text{if } b \text{ then } c_1 \text{ else } c_2 \approx \text{if } \neg b \text{ then } c_2 \text{ else } c_1$$
$$x := 0;\ y := x + 1 \approx y := 1;\ x := 0$$

The proofs are again straightforward.

2.3.5 Recursively Defined Functions

As we remarked earlier, certain types of language construct require more formal machinery to capture their meaning in denotational semantics.

Once such problematic construct is the while loop, which we omitted from our earlier discussion of the language IMP1. We could attempt to define its meaning as follows:

$$\boxed{C[\text{while b do c}] = \underline{\lambda}s.B[b]s \Rightarrow C[\text{while b do c}](C[c]s) \,\square\, s}$$

This certainly captures our intuition about what a while-loop should do, but it is, unfortunately violating the requirement that the meaning of a program phrase must be defined in terms of its proper subphrases.

The theory of domains provides us with a rigorous way of capturing the semantics of while, along with other recursive objects. Essentially, we need to make formal the meaning of a recursive specification. Space does not permit us to do more than summarise this theory, but here are some key results. For more details, consult the book by Schmidt [2].

A *partial ordering* on a set D is a relation $\sqsubseteq$ which is:

a. reflexive: $\forall a \in D.\ a \sqsubseteq a$;
b. antisymmetric: $\forall a, b \in D.\ a \sqsubseteq b$ and $b \sqsubseteq a \supset a = b$; and
c. transitive: $\forall a, b, c \in D.\ a \sqsubseteq b$ and $b \sqsubseteq c \supset a \sqsubseteq c$

A *least element* of D with respect to this ordering is an element $\perp$ such that $\forall a \in D,\ \perp \sqsubseteq a$. If X is a subset of D, then the *least upper bound*, written $\bigsqcup X$, denotes the element of D (if it exists) such that:

a. $\forall x \in X.\ x \sqsubseteq \bigsqcup X$ and
b. $\forall d \in D.$ if $\forall x \in X, x \sqsubseteq d$ then $\bigsqcup X \sqsubseteq d$

A non-empty subset X of D is called a *chain* if, $\forall a, b \in X$ either $a \sqsubseteq b$ or $b \sqsubseteq a$. Finally D is called a *domain* if D has a least element $\perp$ and every chain has a least upper bound.

Suppose that A and B are sets with partial orderings. A function $f : A \rightarrow B$ is called *monotonic* if $\forall x, y \in A,\ f(a) \sqsubseteq f(b)$. The function f is called continuous if it is monotonic and, furthermore, for every chain $X \sqsubseteq A,\ f(\bigsqcup X) \sqsubseteq \bigsqcup\{f(x) \mid x \in X\}$. If $F : D \rightarrow D$ is continuous, then a *fixed point* of F is an element d of D such that $F(d) = d$. It is called the *least fixed point* of f if, $\forall e \in D,\ F(e) = e \supset d \sqsubseteq e$.

The key result of domain theory is the following:

Theorem: If D is a domain, then every function $F : D \rightarrow D$ has a least fixed point, given by:

$$\text{fix } F = \bigsqcup\{F^i(\perp) \mid i \geq 0\}$$

We then take the meaning of a recursive specification f = F (f) to be fix F.

It can be shown that the domain constructors described earlier all construct new domains from given domains (which justifies their name). A primitive domain such as $Nat_\perp$ is given the discrete partial ordering: $a \sqsubseteq b \Leftrightarrow a = b$ or $a = \perp$ (such domains are called *flat*).

Now we are in a position to define the meaning of a while loop as follows:

$$\boxed{C[\text{while b do c}] = \text{fix } (\lambda f.\underline{\lambda}s.B[b]s \Rightarrow f(C[c]s) \,\square\, s\ )}$$

To complete the discussion of fixed points, we note that, for certain kinds of predicates (called *inclusive predicates*), we have an important induction principle, known as *fixed-point induction*. For such a predicate P on a domain D, we have the inference rule:

$$\frac{P(\bot) \quad \forall d \in D,\ P(d) \supset P(F(d))}{P(\text{fix } F)}$$

This result is very important for reasoning about fixed points in theorem provers such as LCF.

2.3.6 Limitations

By now, the methods for giving denotational semantics for sequential programming languages are well-understood: all the constructs used in such languages have been given well-defined mathematical formulations. This is true of blocks, jumps, procedures and functions and so on. However, the method has its limitations [6].

The extension of denotational semantics to cover parallel languages is not straightforward. The problem is that denotational semantics makes heavy use of functions, whereas the essential property of parallel programs is non-determinism. In contrast, operational semantics (see below) extends quite neatly to cover the case of parallel languages, and is, for example, the standard way of formulating Robin Milner's Calculus of Communicating Systems.

Also, the method does not scale up well to real programming languages — whose domains can be extremely complex — and it is fair to say that denotational semantics definitions have not really affected the mainstream development of languages such as Ada [8]. Denotational semantics definitions are also very hard to read, and make little sense to the non-expert.

Another issue is that it is not possible to reuse parts of the description of one language in another language (in other words, denotational semantics has no construction analogous to that of modules in software engineering).

2.4 Operational Semantics

There is today a good deal of interest in the natural or operational style of semantics, which grew out of the work of Gordon Plotkin [9]. His aim was to formalise language semantics in a natural way, in terms of transition systems, without being too worried about mathematical rigour. The method has found favour among those who wish to study real languages without wanting to use a full denotational semantics. In fact, the definition of Standard ML is given in terms of a number of transition rules.

An operational semantics is given by defining a *transition system*, which we define to be a pair $(\Gamma, \Rightarrow)$, where Γ is a set of configurations, and $\Rightarrow$ is a relation on Γ, with the interpretation that $\gamma_1 \Rightarrow \gamma_2$ means that γ_1 evaluates to γ_2. Many systems actually fall into the category of transition system, but we are most interested in the case of language semantics, where a number of inference rules capture the meaning of the language.

Operational semantics has the advantage that it is an easier formalism to understand than denotational semantics. Also, as mentioned above, it is better able to cope with the treatment of non-deterministic languages, where meaning is captured by relations rather than functions. The main disadvantage is that it is tied to the way an interpreter will evaluate phrases in the language, and so it is not a good formalism for reasoning about termination, or loops which may iterate an arbitrary number of times depending on the input.

Our discussion of operational semantics draws on the lecture notes of Milner [10]

2.4.1 Semantics of FUNC1

As an example, we shall consider a very simple language FUNC1, with simple expressions and a mechanism for local scoping, which later will be extended to a functional language.

SYNTAX OF FUNC1

e : expression
d : declaration
x : identifier
n : number

$e ::= x \mid n \mid e_1 + e_2 \mid \text{let } d \text{ in } e$
$d ::= x = e \mid d_1 \,;\, d_2$

Phrases in this language are evaluated in the context of an environment, which records the current bindings of identifiers to their values (in this case numbers). We shall regard an environment as a set of pairs, as follows:

$$E = \{(x_1, n_1), \ldots, (x_k, n_k)\}$$

Our transition system relation has the form:

$$\text{Environment} \vdash \text{phrase} \Rightarrow \text{result}$$

in which the result is a number if the phrase is an expression, and is an environment if the phrase is a declaration. Below are given the inference rules for the operational semantics of FUNC1. We use the notation E_1E_2 to denote the new environment obtained from E_1 by overwriting its bindings with those of E_2:

OPERATIONAL SEMANTICS FOR FUNC1

Expressions :

$$E \vdash x \Rightarrow n \qquad \text{if } (x, n) \in E$$

$$E \vdash n \Rightarrow n$$

$$\frac{E \vdash e_1 \Rightarrow n_1 \quad E \vdash e_2 \Rightarrow n_2 \quad n = n_1 + n_2}{E \vdash e_1 + e_2 \Rightarrow n}$$

$$\frac{E \vdash d \Rightarrow E' \quad EE' \vdash e \Rightarrow n}{E \vdash \text{let } d \text{ in } e \Rightarrow n}$$

Declarations :

$$\frac{E \vdash e \Rightarrow n}{E \vdash x = e \Rightarrow \{(x, n)\}}$$

$$\frac{E \vdash d_1 \Rightarrow E_1 \quad EE_1 \vdash d_2 \Rightarrow E_2}{E \vdash d_1 \,;\, d_2 \Rightarrow E_1E_2}$$

As examples, we have:

$$E \vdash \text{let } x = 5;\ y = 10 \text{ in } x + y \Rightarrow 15$$
$$E \vdash \text{let } y = 3 \text{ in let } x = y + 1 \text{ in let } y = 20 \text{ in } x \ \Rightarrow 4$$

The second example illustrates the fact that we have *static binding*: x uses the value of y at definition-time, not at call-time.

A formal proof of these examples uses the inference rules for the semantics to work backwards from what it is required to prove (the "goal"), constructing a proof tree whose nodes are trivially true.

There is a natural notion of *equivalence* with respect to the semantics for the language FUNC1. We define equivalence of expressions and declarations as follows:

$$e_1 \approx e_2 \equiv \forall E, n: \quad E \vdash e_1 \Rightarrow n \Leftrightarrow E \vdash e_2 \Rightarrow n$$
$$d_1 \approx d_2 \equiv \forall E, E': \quad E \vdash d_1 \Rightarrow E' \Leftrightarrow E \vdash d_2 \Rightarrow E'$$

For example, it is easy to show that:

$$e_1 + e_2 \approx e_2 + e_1$$
$$\text{let } x = 1 \text{ in } x + 3 \approx \text{let } y = 2 \text{ in } y + 1$$
$$x = 7;\ y = 9 \approx y = 9;\ x = 7$$

2.4.2 Semantics of FUNC2

Now let us extend to the language FUNC1 to a more interesting and useful language, called FUNC2. It is a *functional* language: functions are allowed to be values, and can be passed as arguments to other functions. We shall add the following new expressions: conditional expressions, pairing e_1, e_2, function application $e_1\ (e_2)$ and lambda-expressions $\lambda x.e$. We have also included parallel declarations, with the construct "and". Here is the syntax of FUNC2:

SYNTAX OF FUNC2

e : expression

d : declaration

x : identifier

c : constant $= \text{number} + \{tt, ff\} + \{+, \times\}$

$e ::= x \mid c \mid e_1\ , e_2 \mid e_1(e_2)$
$\quad\quad \mid \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \mid \lambda x.e \mid \text{let } d \text{ in } e$

$d ::= x = e \mid d_1\ ;\ d_2 \mid d_1 \text{ and } d_2$

An environment will still associate identifiers with values; however, this time the set of values is defined as follows:

- every constant is a value;
- if v_1 and v_2 are values, then v_1, v_2 is a value; and
- if E is an environment, x an identifier and e an expression, then the *closure* <x,e,E> is also a value.

Function closures (or function values) contain all the information necessary to evaluate a function. we evaluate e (the function body) in the environment E extended by associating the actual parameter value with the formal parameter x. Closures are needed to avoid inconsistencies due to the use of the same identifier as both a bound and a free variable (well-known as the "funarg problem" in LISP).

Now if we wanted to make rigorous sense of such a recursively defined set we would, of course, require the machinery of domain theory. Thus the denotational semantics for this language needs some special care. However, the operational semantics is not difficult to formulate, and is given below:

OPERATIONAL SEMANTICS FOR FUNC2

Expressions :

$$E \vdash x \Rightarrow v \quad \text{if } (x, v) \in E$$

$$E \vdash c \Rightarrow c$$

$$\frac{E \vdash e_1 \Rightarrow v_1 \quad E \vdash e_2 \Rightarrow v_2}{E \vdash e_1, e_2 \Rightarrow v_1, v_2}$$

$$\frac{E \vdash e_1 \Rightarrow \langle x, e, E' \rangle \quad E \vdash e_2 \Rightarrow v \quad E'\{(x, v)\} \vdash e \Rightarrow v'}{E \vdash e_1(e_2) \Rightarrow v'}$$

$$\frac{E \vdash e_1 \Rightarrow c \quad E \vdash e_2 \Rightarrow v \quad \text{apply}(c, v) = v'}{E \vdash e_1(e_2) \Rightarrow v'}$$

$$\frac{E \vdash e_0 \Rightarrow \text{true} \quad E \vdash e_1 \Rightarrow v}{E \vdash \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \Rightarrow v}$$

$$\frac{E \vdash e_0 \Rightarrow \text{false} \quad E \vdash e_2 \Rightarrow v}{E \vdash \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \Rightarrow v}$$

$$E \vdash \lambda x\, e \Rightarrow \langle x, e, E \rangle$$

$$\frac{E \vdash d \Rightarrow E' \quad EE' \vdash e \Rightarrow v}{E \vdash \text{let } d \text{ in } e \Rightarrow v}$$

Declarations :

$$\frac{E \vdash e \Rightarrow n}{E \vdash x = e \Rightarrow \{(x, n)\}}$$

$$\frac{E \vdash d_1 \Rightarrow E_1 \quad EE_1 \vdash d_2 \Rightarrow E_2}{E \vdash d_1 \,;\, d_2 \Rightarrow E_1 E_2}$$

$$\frac{E \vdash d_1 \Rightarrow E_1 \quad E \vdash d_2 \Rightarrow E_2}{E \vdash d_1 \text{ and } d_2 \Rightarrow E_1 E_2}$$

Note that, for constants such as + which happen to be operations, we assume that the semantic algebras support a function **apply**, which evaluates its application to arguments, for example apply (+, (3,6))=9. Also note that there are two rules for function application, according as the operator e_1 of the application evaluates to a closure or a constant value.

2.4.3 Semantics of FUNC3

The next stage is to extend FUNC2 to allow for recursively defined functions in the language. The extension to the syntax is simply to allow a declaration to be qualified by the word "rec". So we extend the syntax of declarations to be:

$$\boxed{\mathrm{d} ::= \mathrm{x} = \mathrm{e} \mid \mathrm{d}_1 \,;\, \mathrm{d}_2 \mid \mathrm{d}_1 \text{ and } \mathrm{d}_2 \mid \mathrm{rec}\ \mathrm{d}}$$

This will allow the local declaration of mutually recursive functions, for example:

$$\text{let rec } (\mathrm{f}_1 = \lambda \mathrm{x}_1.\mathrm{e}_1 \text{ and } \mathrm{f}_2 = \lambda \mathrm{x}_2.\mathrm{e}_2) \text{ in e}$$

in which, typically, both e_1 and e_2 might contain applications of f_1 or f_2.

Milner [10] gives a careful discussion of how the operational semantics needs to be modified to allow for recursion. We shall just give the inference rules which need to be changed.

Firstly, we need to extend the notion of function closure to have the form <x,e,E,E'>, in which the new fourth component is an environment specifying the function identifiers which must be treated recursively when we evaluate the body e of the closure. We modify the rule for lambda-expressions as follows:

$$\boxed{\mathrm{E} \vdash \lambda \mathrm{x}.\mathrm{e} \Rightarrow \langle \mathrm{x}, \mathrm{e}, \mathrm{E}, \emptyset \rangle}$$

The semantics of recursive declarations requires the key notion of unfolding an environment, defined as follows. if E is an environment, then whenever E contains a member

$$\left(\mathrm{f}_i, \left\langle \mathrm{x}_i, \mathrm{e}_i, \mathrm{E}_i, \mathrm{E}_i' \right\rangle\right)$$

then UNFOLD E will contain instead the member

$$(\mathrm{f}_i, \langle \mathrm{x}_i, \mathrm{e}_i, \mathrm{E}_i, \mathrm{E} \rangle)$$

Now we can give the rules for recursive declarations and function applications:

$$\frac{\mathrm{E} \vdash \mathrm{d} \Rightarrow \mathrm{E}'}{\mathrm{E} \vdash \mathrm{rec}\ \mathrm{d} \Rightarrow \mathrm{UNFOLD}\ \mathrm{E}'}$$

$$\frac{\mathrm{E}_0 \vdash \mathrm{e}_1 \Rightarrow \langle \mathrm{x}, \mathrm{e}, \mathrm{E}, \mathrm{E}' \rangle \quad \mathrm{E}_0 \vdash \mathrm{e}_2 \Rightarrow \mathrm{v} \quad \mathrm{E}(\mathrm{UNFOLD}\ \mathrm{E}')\{(\mathrm{x}, \mathrm{v})\} \vdash \mathrm{e} \Rightarrow \mathrm{v}'}{\mathrm{E}_0 \vdash \mathrm{e}_1(\mathrm{e}_2) \Rightarrow \mathrm{v}'}$$

We shall meet the languages FUNC1, FUNC2 and FUNC3 again later on in Chapter 6, where we set up proof procedures for reasoning about these languages.

2.4.4 Semantics of IMP1

It is not hard to give an operational semantics for the language IMP1 (whose denotational semantics was given earlier). This language is essentially that discussed in Milner's notes [10]. It has also been studied by Rachel Roxas and Malcom Newey [11]. We shall not give its semantics in full here.

Phrase evaluation is now written as follows:

$$\text{environment} \vdash \text{expression}, \text{memory}_1 \Rightarrow \text{value}, \text{memory}_2$$
$$\text{environment} \vdash \text{program}, \text{memory}_1 \Rightarrow \text{memory}_2$$

where we have allowed for expression evaluation to have side-effects and change the memory (for the simplest imperative langauges there will be no side-effects). In general, there is also an environment to allow local declarations, as for our functional languages.

Omitting environments for now, we give just the rules for assignment and while loops:

$$\frac{e, M \Rightarrow n, M'}{x := e, M \Rightarrow \text{update}(x, n, M')}$$

$$\frac{b, M \Rightarrow \text{true}, M' \quad p, M' \Rightarrow M'' \quad \text{while } b \text{ do } p, M'' \Rightarrow M'''}{\text{while } b \text{ do } p, M \Rightarrow M'''}$$

$$\frac{b, M \Rightarrow \text{false}, M'}{\text{while } b \text{ do } p, M \Rightarrow M'}$$

It is not difficult to formulate operational semantics for the other constructs found in imperative languages (procedure calls, jumps etc), but we shall not do this here.

2.5 Axiomatic Semantics

The key ideas in axiomatic semantics were put forward in 1969 by C A R Hoare [12] in a paper describing a logic for capturing assertions about a small imperative language. This paper had a major impact, and has the distinction of being one of the most widely cited papers in computer science. There has been tremendous activity in the area over the last twenty years, as witnessed by the extensive review of the subject by [13]. We shall give a brief discussion, following Schmidt [2].

In Hoare logic we deal with partial correctness assertions of the form {P} c {Q}, which specify the behaviour of some command c in terms of predicates P and Q, which are functions of program variables. Informally, the assertion means that "if P is true in some state, and c is then executed and terminates, then Q is true in the resulting state". In the language of denotational semantics, such an assertion reads:

$$\{P\}c\{Q\} \equiv \forall s \in \text{Store}, \quad B[P]s \wedge \neg(C[c]s = \bot) \supset B[Q](C[c]s)$$

Some examples of the Hoare rules for assignments, sequencing and while loops are given below:

$$\{P(e/x)\}\ x := e\ \{P\}$$

$$\frac{\{P\}\ c_1\ \{Q\}\quad Q \supset R\quad \{R\}\ c_2\ \{S\}}{\{P\}c_1 ; c_2\{S\}}$$

$$\frac{P \wedge \{b\}\ c\ \{P\}}{\{P\}\ \text{while } b \text{ do } c\ \{\neg b \wedge P\}}$$

The point we want to make here is that these rules can be shown to follow from the denotational semantics of the language [2].

The Hoare rules are very popular in program verification, and many treatments of the subject take these rules as their starting point. However, the Hoare rules will not be used further in this paper.

Chapter 3 INTRODUCTION TO ML

In this chapter we shall give a brief overview of the programming language ML (ML stands for Meta-Language), on which both the HOL system and Isabelle are built. ML was originally developed during work on the early theorem prover LCF (LCF stands for Logic for Computable Functions), which was developed by Robin Milner and collaborators in the early 1970's [14].

ML has now become an important programming language in its own right. Even during its early stages, the language featured higher order functions, along with a robust type system with an method for type inferencing, and had an exception raising mechanism to facilitate the definition of proof tactics. ML was redesigned to incorporate new ideas from the work of Burstall's group on HOPE and CLEAR , such as pattern matching and the modular construction of specifications using signatures in the interfaces. Most recently, a major enhancement to ML has been that of modules, due to MacQueen.

The agreed standard for the language is called Standard ML (SML), and its definition is given in [15]. A readable, but low-level, introduction to the language is the book by Wikstrom [16]. Another readable summary of ML is the manual for the version under development by Malcolm Newey and collaborators at the Australian National University[17]. However, since we are discussing the HOL system first, the examples in this chapter will be for the slightly different dialect of the language on which HOL is built. (The theorem prover Isabelle is built on SML). We do this for convenience: the differences are not significant for the examples we wish to give.

3.1 Features of ML

ML is a modern functional programming language, with a number of powerful features which follow modern software engineering principles. In fact ML is the paradigm for a functional language which we have in mind for our experiments on the verification of programs written in functional languages. Here are the highlights of ML.

- □ It is a functional language — functions are first-class objects and can be passed as arguments to other functions, or returned as values.
- □ It is statically-scoped: identifiers are associated with values according to where they appear in the program text (and not on the run-time behaviour of the program). This is safer than dynamic scoping (as in LISP).
- □ It is a strongly typed language — every ML expression has a statically-determined type. The type of an expression can usually be inferred automatically, by an algorithm due to Milner. This catches many trivial errors at compile-time, and promotes good programming practice.
- □ It is polymorphic — type expressions may contain type variables, to allow for functions to be defined on a class of arguments of different types.
- □ It has facilities for abstraction — the user can define new abstract data types and hide the details of their implementation from functions which make use of them.
- □ It has a modules facility, allowing the grouping of large ML programs into separate units which can be separately compiled.
- □ It has an exception trap mechanism, to allow the uniform handling of user and system-generated exceptions.
- □ It has a rigorous semantics — the language definition of SML [15] is expressed in terms of operational semantics, so that implementors and others know what is required. There is also an (unpublished) denotational semantics for the language, due to Gordon and Milner.

3.2 Syntax

In this section we shall summarise the syntax of ML, concentrating on the bare essentials. The following table shows which variables are used to range over the various semantic domains of ML.

Table 1 Notation for Semantic Domains

Variable	Ranges over
var	variables
con	constructors
constexp	constant expressions
d	declarations
b	bindings
p	patterns
e	expressions

A simplified syntax of ML, in which we omit types, precedence information, constructs which are equivalent to others, and certain exotic forms of exceptions can be summarised as follows:

$$\begin{aligned}
d &::= \text{let } b \mid \text{letref } b \mid \text{letrec } b \\
b &::= p = e \mid \text{var } p_1\ p_2 \ldots p_n\ = e \mid b_1 \text{ and } b_2 \text{ and } \ldots \text{ and } b_n \\
p &::= \ \mid \text{constexp} \mid \text{var} \mid \text{con } p \mid p_1 . p_2 \mid p_1 , p_2 \mid [] \mid [p_1 ; \ldots ; p_n] \\
e &::= \text{constexp} \mid \text{var} \mid \text{con} \mid e_1\ e_2 \\
&\quad \mid \text{px } e \mid e_1 \text{ ix } e_2 \mid p\ := e \mid \text{failwith } e \\
&\quad \mid \text{if } e_1 \text{ then } e_2\ \{\text{else } e_3\} \\
&\quad \mid e_1 \mathbin{?} e_2 \\
&\quad \mid \text{while } e_1 \text{ do } e_2 \\
&\quad \mid e_1 ; \ldots ; e_n \mid [] \mid [e_1 ; \ldots e_n] \mid d \text{ in } e \\
&\quad \mid \backslash p_1\ p_2 \ldots p_n . e \\
&\quad \mid \text{fun } p_1 . e_1 \mid p_2 . e_2 \mid \ldots \mid p_n . e_n
\end{aligned}$$

3.3 Examples

Rather than give a formal definition of the semantics of ML, we shall be content to explain the language by means of a number of examples.

ML constructs are evaluated in the context of an environment and a store (these concepts have already been discussed in 2). The environment holds current bindings. It specifies what the variables and constructors in use denote — they may be bound either to *values* or *locations*. The store specifies the contents of locations (which must be values).

Evaluation of a declaration d changes the bindings in the environment of the identifiers declared in d. Evaluation of an expression yields a value. Any assignment done during an evaluation changes the store (such changes are called side-effects).

Expressions and patterns may optionally be given types (for example `x:int`), which forces the type-checker to assign an instance of the asserted type to this construct.

We shall illustrate the various constructs in ML by means of some simple sessions with the HOL system. In these interactions, # is the HOL prompt, and the user enters ML phrases followed by two semi-colons.

3.3.1 Expressions

Expressions can take a number of forms. A simple example is:

```
#5 + 7;;
12 : int
```

The value of the expression (12) is returned, along with its type (int). Here are some other examples:

```
#'this is a string';;
'this is a string' : string

#('hello', 9 - 5);;
('hello', 4) : (string * int)

#if true then 3 else 4;;
3 : int

#if (5=7) then 'yes' else 'no';;
'no' : string
```

3.3.2 Declarations

The declaration `let x = e` evaluates `e` and binds the resulting value to the identifier x. For example:

```
#let x = 7;;
x = 7 : int


#let y = x - 3;;
y = 4 : int


#let x = 10 and y = 7;;
x = 10 : int
y = 7 : int


#let x = 2 in x + 2;;
4 : int
```

3.3.3 Functions

The general form of a function definition is `let f x = e`, where x is a formal parameter, and e the body of the function. For example we can define the successor function as follows:

```
#let succ x = x + 1;;
succ = - : int -> int


#succ 7;;
8 : int
```

Note that the type inferencing mechanism of ML means that the type of `succ` has been inferred to be `int -> int` without having to declare x explicitly to be of type `int`.

Functions of several arguments can be defined:

```
#let add x y = x + y;;
add = - : (int -> int -> int)


#let add3 = add 3;;
add3 = - : (int -> int)


#add3 5;;
8 : int
```

An equivalent notation for functions is the lambda-expression. A backslash '\' is used to approximate a lambda. The expression `let f = \x.e` is equivalent to `let f x = e`. Recursive functions may be defined using the keyword letrec:

```
#let f = \x.x+1;;
f = - : (int -> int)

#f 7;;
8 : int

#letrec fact n = if n=0 then 1 else n*fact(n-1);;
fact = - : (int -> int)

#fact 5;;
120 : int
```

3.3.4 Lists

All the elements of a list must be of the same type:

```
#[1;2;3;4;5];;
[1; 2; 3; 4; 5] : int list
```

We have the following standard operations on lists:

```
#hd [1;2;3];;
1 : int

#tl [1;2;3];;
[2; 3] : int list

#'one'.['two'; 'three'];;
['one'; 'two'; 'three'] : string list

#[1;2;3] @ [4;5;6];;
[1; 2; 3; 4; 5; 6] : int list
```

3.3.5 Polymorphism

Let us inspect the type of the function `hd`:

```
#hd;;
- : (* list -> *)
```

This says that `hd` has many types: it is defined on any list of elements with the same (but arbitrary) type, (hence the type variable *), and returns an element of that type. Such functions are called polymorphic.

3.3.6 Failure

Some standard functions fail at run-time on certain arguments:

```
#1/0;;
evaluation failed     div

#hd(7);;

ill-typed phrase: 7
has an instance of type  int
which should match type  * list
1 error in typing
typecheck failed
```

Failures can be trapped with ?, so the value of the expression `e1 ? e2` is `e1` unless `e1` causes a failure, in which case the result is the value of `e2`. A failure may be forced as follows:

```
#failwith 'mistake!';;
evaluation failed     mistake!
```

3.3.7 New Types

ML is very flexible in this respect. We may simply abbreviate a type, as in:

```
#lettype stringpair = string # string;;
type stringpair defined
```

or define a new type, for example:

```
#type card = ace | king | queen | jack | other of int;;

New constructors declared:
ace : card
king : card
queen : card
jack : card
other : (int -> card)
```

which declares a new type consisting of four constructors: the constants `king`, `queen` and `jack`, and the function `other`. A function whose argument is of such a type is defined by the general expression

$$\mathrm{fun}\ \mathrm{pat}_1.e_1 \ |\ \mathrm{pat}_2.e_2 \ |\ \ldots \ |\ \mathrm{pat}_n.e_n$$

Such an expression denotes a function which, given a value v, selects the first pattern which matches v, say pat_i, binds the variables of pat_i to the corresponding components of the value, and then evaluates the expression e.

So, for example, a function giving the standard value of a card in contract bridge is given by:

```
#let value = fun ace        . 4
               | king       . 3
               | queen      . 2
               | jack       . 1
               | (other n)  . 0;;
#value = - : (card -> int)
```

Then the total value of a hand of cards (represented as a list H) is given by:

```
#letrec totalvalue H =
   if null H then 0 else value(hd(H)) + totalvalue (tl(H));;
#totalvalue = - : (card list -> int)


#totalvalue [ace;king;king;jack;(other 5)];;
11 : int
```

Recursive types can be defined: for example, we could define the positive integers by:

```
#rectype int = zero | succ of int;;

New constructors declared:
zero : int
succ : (int -> int)
```

New types can also be defined by abstraction. For example here is a definition of an abstract type called `set`, constructed from a list of integers. Note that the internals of this type are hidden from the user:

```
#abstype set = int list
with make_set = \x. abs_set x and
empty = abs_set [] and
size = \s. length (rep_set s);;
###make_set = - : (int list -> set)
empty = - : set
size = - : (set -> int)
#let s = make_set [1;2;3;4;5];;
s = - : set

#size s;;
5 : int

#size empty;;
0 : int
```

3.3.8 Imperative Features

We have relegated until last those features of ML which lie strictly outside of functional languages. Assignable variables are created with the keyword `letref`. The `while` construct is also avilable.

```
#letref a = 3;;
a = 3 : int

#a;;
3 : int

#a := 7;;
7 : int

#a;;
7 : int
let fact n = letref count = n and result = 1 in
while count > 0 do count, result := count-1, count*result; result;;
#fact 5;;
120 : int
```

Chapter 4 THE HOL SYSTEM

4.1 Introduction

In this chapter we shall give an overview of HOL, a system developed by Mike Gordon at the University of Cambridge. HOL supports interactive theorem proving in higher order logic. It inherits many ideas from LCF [14], a good up-to-date account of which may be found in the book by Larry Paulson [18]. As in LCF, the language ML provides the environment in which terms and theorems of the logic can be denoted and theorem proving takes place.

HOL is really a proof-assistant and proof checker. It will not prove complex theorems automatically: the user must have an idea of the way the proof will work, and apply the appropriate steps (called tactics) in the proof, which works in a goal-directed fashion. The HOL system manages the proof, taking care of the details of primitive proof steps, and provides a sound theorem proving environment — i.e. the user is assured that a theorem, once obtained, is true within the logic.

HOL provides a natural and highly expressive way of specifying and reasoning about models of abstract systems. It was originally suggested as a tool for the verification of hardware, and it is fair to say that most of the activity in HOL is in hardware (with Mike Gordon's group at Cambridge primarily involved in this area). Nevertheless, HOL has been applied to other areas, including protocol verification [19], mathematical theories such as groups and integers, machine architecture specification [20], security policy modelling [21]. The HOL TUTORIAL [22] gives several examples. HOL is just beginning to be applied to the area of software verification, which is the concern of this paper.

The following is not meant to be exhaustive, but rather to give a flavour of working with HOL, as well as highlighting both those features of the system which support reasoning about programming languages, and those which hinder such reasoning.

4.2 Higher Order Logic

The HOL logic is a version of *higher order logic* based on Church's formulation of simple type theory [23]. It is a variant of typed polymorphic λ–calculus, with formulae being identified with terms of boolean type. Variables can range over functions and predicates, and functions can take other functions as arguments (hence 'higher order'). The important property of t

he HOL logic is that it is expressive enough to be able to formulate mathematical theories: one might think of it as being as formal tool which replaces the usual mathematician's meta-language of informal description and proof with a formal system which captures the same ideas.

An important notion is that of a **theory**, which is a collection of types, constants, definitions and axioms — as with any logical system — but which also contains an explicit list of theorems which have already been proved from the axioms and definitions, and perhaps earlier theorems. A HOL theory is a dynamic object which can be extended or even modified during an interaction with the HOL system.

4.2.1 Types

The HOL logic is typed; we can think of types as expressions that denote sets. We shall use the generic variable σ to range over arbitrary types. The possible kinds of types are given by the following table adapted from [22]:

Table 2 HOL Types

Kind of Type	HOL Notation	ML Notation	Description
Type variable	α	":* ..... "	arbitrary type
Type constant	c	":op "	fixed type
Function type	σ -> σ'	":σ -> σ' "	functions from σ to σ'
Compound type	(σ_1, ..., σ_n) op	":(σ_1, ..., σ_n) op"	general type constructor

Type variables denote arbitrary (non-empty) sets, and are used to specify ranges of types in the logic. **Type constants,** or **atomic types** denote fixed sets of values. Each theory determines some collection of type constants. For example, the standard constant type **bool** denotes the set of truth values. The **function type** $\sigma \rightarrow \sigma'$ denotes the set of total functions from the set denoted by σ to the set denoted by σ'. Finally, the **compound type** (σ_1, ..., σ_n) op gives a general means of constructing new types, for example the product type $\sigma_1 \times \sigma_2$ = (σ_1, σ_2) prod. Types containing type variables will be called **polymorphic**, all other types being **monomorphic**. An **instance** σ' of a type σ is obtained by replacing all the occurrences of a type variable in σ by a type.

Also shown in the table is the representation in ML of the various kinds of type. This will be discussed later on.

4.2.2 Terms

The terms of the HOL logic are simply expressions denoting elements of the sets denoted by types. We shall use the variable t to range over terms. The following table summarises the four kinds of terms in the logic:

Table 3 Primitive Terms of the HOL Logic

Kind of term	HOL Notation	ML Notation	Description
Variable	x	"var : σ "	variable var of type σ
Constant	c	"const : σ "	constant of type σ
Application	t t'	"t t' "	function t applied to t'
Abstraction	λx.t	"\x. t "	lambda expression

A **function application** t t' denotes the result of applying the function denoted by t to the value denoted by t', while the **λ−term** λx.t denotes the function v → t [v/x], where t [v/x] denotes the result when v is substituted for x in t.

Although the terms just given make no mention of types, each term in the logic actually has a unique type. If we need to be explicit, we write t_σ to express the fact that t is of type σ. Once again, we call a term containing a type variable **polymorphic**. Any term input to the system must be well-typed according to the rules of the logic. HOL has a type checker for logical terms based on the ML type checking algorithm.

4.2.3 Logical Formulae

Every theory is assumed to contain the constant type **bool**; it is an important feature of the HOL logic that logical formulae are then identified with terms of type **bool**. The HOL system does not distinguish between them at all[1]. also, various logical constructs are assumed to be present in each theory. We shall

[1] In Isabelle's version of Higher Order Logic, terms of type bool are distinguished from formulae, but are interchangeable by means of certain built-in equivalences.

not go into details, but just assume that the logic is expressive enough in that it contains the following constructs:

Table 4 Derived Logical Constructs of HOL

Kind of term	HOL Notation	ML Notation	Description
Truth	T	"T "	true
Falsity	F	"F "	false
Negation	$\neg t$	"~t "	not t
Disjunction	$t \vee t'$	"t V t' "	t or t'
Conjunction	$t \wedge t'$	"t Λ t' "	t and t'
Implication	$t \supset t'$	"t ==> t' "	t implies t'
Equality	t=t'	"t = t' "	t equals t'
Universal Quantification	$\forall x.t$	"!x.t "	for all x : t
Existential Quantification	$\exists x.t$	"?x.t "	there exists an x such that t
Unique Existential Quantification	$\exists !x.t$	"?!x.t"	there exists a unique x such that t
ϵ-term	$\varepsilon x.t$	"@x.t "	an x such that t
Conditional	$t \Rightarrow t' \mid t''$	"t => t' | t'' "	if t then t' else t''

4.2.4 Constant Definitions

The HOL logic provides ways of introducing definitions in a manner which preserves consistency of the logic. A **constant definition** over some theory is an equation of the form $c_\sigma = t_\sigma$,where

1. c is not already a constant in the theory;
2. t is a closed term (i.e. has no free variables); and
3. all the type variables occurring in t_σ occur in σ.

4.2.5 Deduction and Proofs

The HOL logic is based on **natural deduction**. Sentences of the logic are **sequents**, denoted generally by $\Gamma \vdash t$, where Γ is a set of boolean terms called **assumptions**, and t is a boolean term called the **conclusion**. If Γ is empty, we write simply $\vdash t$.

A **deductive system** Ω consists of a set of inference rules, which we write in the following natural style:

$$\frac{\Delta \vdash t_1 \ \ldots \ \Delta \vdash t_k}{\Delta \vdash t}$$

We read this rule as saying that, if the sequents $\Delta \vdash t_1$, ..., $\Delta \vdash t_k$ all hold, then we can conclude the truth of $\Delta \vdash t$.

We say that a sequent $\Gamma \vdash t$ follows from a set of sequents Δ by a deductive system Ω if there are sequents $\Gamma_1 \vdash t_1$, ..., $\Gamma_n \vdash t_n$ such that

$\Gamma \vdash t = \Gamma_n \vdash t_n$, and
if $1 \leq i \leq n$:
either $\Gamma_i \vdash t_i \in \Delta$, or
$\Gamma_i \vdash t_i$ follows by means of Ω from
members of $\Delta \cup \{\Gamma_1 \vdash t_1, \ldots, \Gamma_{i-1} \vdash t_{i-1}\}$

The sequence $\Gamma_1 \vdash t_1$, ..., $\Gamma_n \vdash t_n$ is then called a **proof** of $\Gamma \vdash t$ from Δ with respect to Ω.

4.2.6 The HOL Deductive System

We shall now give the eight basic rules of inference of HOL's deductive system[2]:

[2] There are also five axioms — which are usually defined by definitional extension of a basic theory and will not be given here

Table 5 Basic Rules of Inference for HOL

Description	Inference Rule
Assumption Introduction	$\overline{t \vdash t}$
Reflexivity	$\overline{\vdash t = t}$
Beta-Conversion	$\overline{\vdash (\lambda x.t_1)t_2 = t_1[t_2/x]}$
Substitution	$\dfrac{\Gamma_1 \vdash t_1 = t_1' \quad \ldots \quad \Gamma_n \vdash t_n = t_n' \quad \Gamma \vdash t[t_1. \ldots, t_n]}{\Gamma_1 \cup \ldots \cup \Gamma_n \vdash t\left[t_1', \ldots, t_n'\right]}$
Abstraction	$\dfrac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash (\lambda x.t_1) = (\lambda x.t_2)}$
Type Instantiation	$\dfrac{\Gamma \vdash t}{\Gamma \vdash t[\sigma_1, \ldots, \sigma_n/\alpha_1, \ldots, \alpha_n]}$
Discharging an Assumption	$\dfrac{\Gamma \vdash t_2}{\Gamma - \{t_1\} \vdash t_1 \supset t_2}$
Modus Ponens	$\dfrac{\Gamma_1 \vdash t_1 \supset t_2 \quad \Gamma_2 \vdash t_1}{\Gamma_1 \cup \Gamma_2 \vdash t_2}$

These inference rules are natural and simple ones to write down, although the side conditions on the variables involved can be rather complicated (for more details see the manual).

4.2.7 Theories

A **theory** in the HOL logic is a quadruple:

$$T = (\mathrm{Struct}_T, \mathrm{Sig}_T, \mathrm{Axioms}_T, \mathrm{Theorems}_T)$$

where

i. Struct_T is the type structure of T;
ii. Sig_T is its signature (its basic constants);
iii. Axioms_T is a collection of sequents; and
iv. $\mathrm{Theorems}_T$ is a set of sequents in which every member follows from Axioms_T by means of the HOL deductive system.

There is a natural notion of **extension** of a theory: a theory T' is an extension of T if:

i. $\mathrm{Struct}_T \subseteq \mathrm{Struct}_{T'}$;
ii. $\mathrm{Sig}_T \subseteq \mathrm{Sig}_{T'}$;
iii. $\mathrm{Axioms}_T \subseteq \mathrm{Axioms}_{T'}$;
iv. $\mathrm{Theorems}_T \subseteq \mathrm{Theorems}_{T'}$;

If $c_\sigma = t_\sigma$ is a constant definition over T, then the **definitional extension** of T by $c_\sigma = t_\sigma$ is the theory

$$T_{+def}(c_\sigma = t_\sigma) = (\mathrm{Struct}_T, \mathrm{Sig}_T \cup c_\sigma, \mathrm{Axioms}_T \cup \{c_\sigma = t_\sigma\}, \mathrm{Theorems}_T)$$

The crucial property of this extension is that it is consistent if the original theory T is consistent.

4.3 The HOL Logic in ML

Having looked a little at the HOL logic, we now need to discuss its representation in ML. It is impossible to go into all the details here, so we shall just go over some key points, referring to the manual for further details.

In the HOL system, the types of HOL terms have the ML type called `type`, while terms of the logic have the ML type `term`. They can be input to HOL enclosed in quotation marks; their explicit form is shown in the tables given in the previous section. Various ML functions exist for creating and manipulating these values. A theorem is represented in HOL by a value of ML abstract type `thm`. The system represents inference rules by ML functions whose arguments are of type `thm` and which return a result of type `thm`. HOL contains a large number of derived theorems and inference rules, which are built up from the basic axioms and primitive inference rules of the logic.

The system is *sound* in that the only way to obtain theorems is by generating a proof. This is done by applying the ML functions representing inference rules, either to axioms or previously generated theorems.

4.3.1 ML Functions for Handling Theories

A theory is represented in the HOL system as a Lisp file called "name.th", with 'name' being the name of the theory supplied as an ML string. Theory files have a hierarchical structure which represents sequences of extensions of an initial theory, which is called HOL. A theory will generally have one or more *parents*, of each of which it is an extension. A session with the HOL system consists of creating a new theory by extending existing theories with a number of definitions (and perhaps axioms). There are two modes of interaction with HOL: in *draft mode*, a theory can be arbitrarily extended, but in *proof mode* only new theorems can be proved.

Here is a summary of the most important ML theory functions:

Table 6 ML Theory Functions

Function	Description
new_theory 'name'	Go into draft mode for a new theory called 'name'
new_parent 'name'	Make 'name' a parent of the current theory
new_type n 'op'	Make op a new n-ary type operator in the current theory
new_constant ('c',σ)	Make a new constant of generic type σ in the current theory
new_axiom ('c',t)	Declare the sequent $\vdash$ t to be an axiom of the current theory with name c.
save_thm ('c',th)	Save the theorem th with name c in the current theory file
close_theory	Save the current theory and exit draft mode
new_definition ('a', "c v_1, ..., v_n = t")	Extends the current theory with a constant definition: declares the sequent $\vdash c = \lambda v_1 \ldots v_n.t$ to be a constant definition called a.
extend_theory 'name'	Go into draft mode for 'name'
load_theory 'name'	Go into proof mode for 'name'
include_theory 'name'	Make all the axioms, theorems, definitions from the theory called 'name' available.

When the HOL system is started up, the initial theory is called HOL. This theory has a complicated ancestry, whose exact structure is not important. What the user needs to know is that there are a certain number of built-in theories, which capture a large body of mathematical knowledge. These theories are : **bool, ind, num, prim_rec, arithmetic, list, tree, combin, ltree, tydefs, sum and one.** The HOL system also has as a set of useful library theories (such as **sets, string, integer** etc) which can be called upon at will.

4.3.2 The Type Definition Package

In our table of theory functions earlier, we deliberately omitted a function called `new_type_definition`, which allows a new type or type-operator to be added to the theory in a conservative (i.e. consistent) fashion. This usually involves a lot of work. For many kinds of types, this function has been superseded by the new type definition package (due to Tom Melham). This package automates the considerable proof effort required to define new concrete (possibly recursive) types. Because of its usefulness for the study of language semantics, we shall look at it in some detail.

The main ML function in the package is

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

where the first string is a name under which we want the result to be stored in the theory, and the second string is a specification of the type, given in a manner rather like ML compound types. It is of the form:

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

where each ty_i^j is either a type expression already defined as a type in the current theory (which must not contain op) or is the name op itself. For example, we could define natural numbers by

```
let nat_Axiom = define_type `nat_Axiom`
      `nat = Z | Suc nat`;;
```

In this case, Z stands for zero, and Suc is the successor function. The theorem returned (nat_Axiom) is just the primitive recursion theorem for the natural numbers.

The type package makes it easy to define recursive functions on these new types. For example, we can define the function `parity` on elements of our type `nat` by

```
new_recursive_definition false nat_Axiom `parity`
  "(parity Z = 0) /\
   (parity (Suc k) = 1 - parity (k))";;
```

When this is input to HOL, the package automatically proves the existence of the primitive recursive function (in this case `parity`), and declares a new constant in the current theory with the above definition as its specification.

The type package also gives us a number of other useful theorems automatically, including an induction theorem for the concrete type, a cases theorem, a theorem stating that the constructors are one to one, and so on.

4.4 Goal Directed Proof

It is possible to carry out proofs in HOL in a forwards manner, starting from known theorems and repeatedly applying inference rules until the required result is obtained.

In practice, however, it is very awkward to do proofs this way, and proofs are almost always not carried out forwards, but in a more natural goal-directed fashion invented by Robin Milner for LCF. We begin with the goal, and then try to reduce the goal to a number of subgoals, whose validity implies that of the original goal. These subgoals are successively decomposed, until eventually we reach known facts.

4.4.1 Tactics and Tacticals

To implement this idea in LCF, Milner invented the notion of tactics. A *tactic* is an ML function which, when applied to a *goal*:

1. reduces a goal to a list of subgoals, and
2. provides a "proof function" which justifies why solving the subgoals will solve the goal.

In ML, we have the following type abbreviations:

```
tactic   = goal -> subgoals
goal     = term list # term
subgoals = goal list # proof
proof    = thm list -> thm
```

We say that a tactic *solves* a goal if it reduces the goal to the empty list of subgoals.

Tactics are specified as follows:

$$\frac{\text{goal}}{\text{goal}_1 \quad \text{goal}_2 \quad \ldots \quad \text{goal}_n}$$

HOL has a rich supply of tactics. For example, we have the tactic CONJ_TAC:

$$\frac{A \wedge B}{A \qquad B}$$

which expresses the fact that, if we want to prove the formula $A \wedge B$, it suffices to prove the subgoals A and B, because we know that from $\vdash A$ and $\vdash B$ we can deduce the theorem $\vdash A \wedge B$. The inference rule which does this is called CONJ.

It is interesting to look at the ML code for CONJ_TAC:

```
let CONJ_TAC : tactic (asl, w) =
 let l,r = dest_conj w in
            ([(asl,l);(asl,r)],\[th1;th2].CONJ th1 th2)
 ? failwith `CONJ_TAC`;;
```

The code shows how the subgoals are constructed, how the proof function is built using CONJ, and what happens if the tactic fails. Even such a simple example also shows how painful the writing of tactics as ML procedures can be!

In practice, new tactics are usually built using ML functions called *tacticals*. An example of a tactical is the sequencing tactical THEN: if T_1 and T_2 are tactics, T_1 THEN T_2 is a tactic which first applies T_1 to the goal, and then applies T_2 to each resulting subgoal. Another impor'ant tactical is ORELSE : T_1 ORELSE T_2 is a kind of "choice" tactic which applies T_1 to the goal, unless it fails, in which case it applies T_2. The tactical REPEAT is for iteration : REPEAT T keeps applying T until it fails.

Becoming a HOL expert means becoming familiar with a range of tactics, and the situations where they can be applied. By using tacticals (which we might also term ***strategies***), we can build quite powerful and sophisticated tactics, tailor-made for the problem domain being studied.

4.4.2 The Subgoal Package

In order to allow interactive proof to be carried out, the HOL system is provided with a subgoal package (along the lines of LCF's) which takes care of proof management. It traverses the tree of subgoals depth-first. The current goal can be expanded into subgoals, which are kept on a goal stack. Once a tactic solves a subgoal, the package automatically applies the appropriate proof functions to compute part of the proof, and then and shows the next subgoal to be proved. Unfortunately, the subgoal package does have its limitations, but is to be improved in future versions of HOL.

Here is an example of a rather artificial but simple interactive proof. First we set the goal to be proved using the function g :term -> void:

```
#g "(!x y z:*.HD [x;y;z] = x) /\ (?n:num. ~(n = 0))";;
"(!x y z. HD[x;y;z] = x) /\ (?n. ~(n = 0))"
```

This goal is expanded into two subgoals with CONJ_TAC:

```
#expand CONJ_TAC;;
OK..
2 subgoals
"?n. ~(n = 0)"

"!x y z. HD[x;y;z] = x"
```

Let us examine the first (bottom) subgoal, which is an obvious assertion about lists of arbitrary elements. To prove this subgoal, we simply rewrite with the definition of the head of a list:

```
#HD;;

|- !h t. HD(CONS h t) = h

#expand(REWRITE_TAC [HD]);;

OK..
goal proved
|- !x y z. HD[x;y;z] = x
```

```
Previous subproof:
"?n. ~(n = 0)"
```

HOL now displays the remaining subgoal, which states the (again obvious) fact that there is some number distinct from zero. We only have to find a suitable n: the value 1 = SUC 0 will do. We expand the goal with EXISTS_TAC:

```
#expand (EXISTS_TAC "SUC 0");;
OK..
"~(SUC 0 = 0)"
```

This goal is solved immediately by rewriting with the derived theorem NOT_SUC:

```
#NOT_SUC;;
|- !n. ~(SUC n = 0)

#expand (REWRITE_TAC [NOT_SUC]);;
OK..
goal proved
|- ~(SUC 0 = 0)
|- ?n. ~(n = 0)
|- (!x y z. HD[x;y;z] = x) /\ (?n. ~(n = 0))

Previous subproof:
goal proved
```

The above proof looks even simpler if we use tacticals (THENL is like THEN but can apply different tactics to the resulting subgoals).

```
#g "(!x y z:*.HD [x;y;z] = x) /\ (?n:num. ~(n = 0))";;
"(!x y z. HD[x;y;z] = x) /\ (?n. ~(n = 0))"
#e (CONJ_TAC THENL [REWRITE_TAC [HD];
EXISTS_TAC "SUC 0" THEN REWRITE_TAC [NOT_SUC]]);;
#OK..
goal proved
|- (!x y z. HD[x;y;z] = x) /\ (?n. ~(n = 0))

Previous subproof:
goal proved
```

There are a number of functions provided for interaction with the subgoal package, the most important of which are summarised in the following table:

Table 7 Subgoal Package Commands

Function	Description
g t	initialises the subgoal package with a new goal
expand (or e)	applies a tactic to the top goal on the stack
backup (or b)	backs up to the previous proof state
rotate (or r)	rotates the order of subgoals on the stack
print_state (or p) n	displays n levels of the goal stack
get_state	returns the current goal stack
set_state s	resets the goal stack to s

Chapter 5 PROGRAM VERIFICATION IN HOL

5.1 Introduction

Whatever our predisposition may be about the best way of describing the semantics of programming languages, it is often the case that a given automated theorem prover will make it more natural and easier for us to adopt one particular method. This is perhaps somewhat surprising.

The early theorem prover LCF was, of course, devised with domain theory (and denotational semantics) in mind. LCF has as built-in constructs the notions of partial orderings, bottom elements, continuity etc. LCF turns all sets into domains, with artificially added bottom elements if necessary; all functions are to be continuous. LCF is a slow system. and is apparently no longer available or supported, owing to the rise of interest in HOL.

Now HOL's logic is not directly tailored to reasoning about programs; it is a general purpose logic, powerful enough to express mathematical theories. The logic does away with LCF's annoying habit of lifting sets like Nat (the natural numbers). Therefore, where possible, proofs of such laws as the associative law for arithmetic

$$x + (y + z) = (x + y) + z \quad \forall x, y, z \in \mathrm{Nat}$$

can be proved without having to reason about cases such as $x = \bot$. This makes using HOL a lot easier. However, the price one pays for this is the need to import the relevant parts of domain theory as, and when, they become necessary.

Despite these considerations, we claim that HOL is useful for formulating denotational semantics definitions, and reasoning about program correctness, for the following reasons:

1. *syntax specifications in* BNF form are easily modelled in HOL by establishing new compound (possibly recursive) data types using the type package;
2. common semantic algebras (for example natural numbers, strings, lists) are either already present in HOL or its libraries;
3. HOL's logic is expressive enough to capture the semantic equations, with the meaning of a phrases being, in general, a recursively defined function on the new data types defined above;
4. assertions about program phrases are easily expressed in HOL; and
5. many tedious proofs in denotational semantics can be taken care of by simple rewriting.

What about operational semantics definitions? We claim that HOL seems less natural here. Consider, for example, the typical rules for constants and for the sum of two expressions:

$$E \vdash v \Rightarrow v$$

$$\frac{E \vdash e_1 \rightarrow v_1 \quad E \vdash e_2 \rightarrow v_2 \quad v = v_1 + v_2}{E \vdash e_1 + e_2 \rightarrow v}$$

which we might formulate as the following HOL axioms:

```
    let NUMBER = new_axiom ('NUMBER', "seq (E, Const v, v)");;
    let SUMMATION = new_axiom ('SUMMATION',
        "!v1 v2. (seq (E, e1, v1) /\ seq (E, e2, v2) /\ (v = v1 + v2
                  ==> seq (E, Plus e1 e2, v)" );;
```

Suppose our goal is "`seq(E,Plus(Const 5)(Const 7),5 + 7)`". We could solve it by using `MATCH_MP_TAC` and the axiom for `SUMMATION`, but then we have the variables `v1` and `v2` which must be instantiated "by hand". Thus the tactic needed to solve the goal depends on the explicit form of the goal. Such tactics are cumbersome to write. What we really want is to be able to treat `v1` and `v2` as scheme variables to be instantiated automatically, and propagate the instantiations to the other subgoals. However, HOL does not have a mechanism for doing this.

It might be more logical to capture the operational semantics rules as new inference rules to be added to HOL. We would then have to write appropriate tactics for them directly in ML, which is messy.

However, other authors have had some success with operational style semantics in HOL. Rachel Roxas and Malcolm Newey [11] have used the method to reason about program transformation for a small imperative language. Also, Version 1.13 of HOL is to be provided with some simple tools for the study of structured operational semantics.

We note here that Mike Gordon [24] has had some success in implementing axiomatic style semantics in HOL. He has constructed the Hoare rules along with corresponding tactics for generating verification conditions. These rules are proved within HOL directly from the denotational semantics of the language.

In this chapter, we shall c ncentrate on capturing denotational semantics definitions within HOL.

5.2 The Language IMP1

In this section we implement the imperative language IMP1, whose denotational semantics was given in Chapter 2.3. To begin with, we declare a new theory called **imp1**, with **string** as a parent (to handle identifiers). We also load some useful inference rules for strings and some special tactics. Note: in the following, HOL's responses will be omitted — unless they provide useful information.

```
#system `rm imp1.th`;;
new_theory `imp1`;;
load_library `string`;;
```

5.2.1 Semantic Algebras

It is straightforward to formalise the semantic algebras for IMP1 in HOL. Identifiers may be modelled by strings

```
#new_type_abbrev (`Identifier`, ":string");;
```

The domains Tr (truth values) and Nat (natural numbers) are already present in HOL as the types :bool and :num. However, we shall need also the lifted domain $\mathrm{Nat}_{\perp}$, which we express as

```
#let Number_Axiom = define_type `Number`
  `Number =  number num |  undef_num`;;
#Number_Axiom =
 - !f e. ?! fn. (!n. fn(number n) = f n) /\ (fn undef_num = e)
```

Here we have used the type package to construct this domain, with undef_num being the undefined element. HOL returns the primitive recursion theorem for this compound type. A very useful feature of the type package is that it can prove a number of theorems about new types automatically. We have an induction theorem:

```
#let Number_Induct = prove_induction_thm Number_Axiom;;
Number_Induct = |- !P. (!n. P(number n)) /\ P undef_num ==> (!N. P N)
```

We also have theorems which state that the constructor functions are one-to-one and distinct, and a theorem which permits case analysis:

```
#save_thm (`Number_one_one`,
   prove_constructors_one_one Number_Axiom);;
|- !n n'. (number n = number n') = (n = n')

save_thm (`Number_distinct`,
   prove_constructors_distinct Number_Axiom);;
```

```
|- !n. ~(number n = undef_num)

save_thm ('Number_cases',
   prove_cases_thm Number_Induct);;
|- !N. (?n. N = number n) \/ (N = undef_num)
```

The functions is_number and is_undefined act as discriminators, while get_num retrieves the number from a proper element of the domain.

```
#let IS_NUMBER_DEF = new_recursive_definition false Number_Axiom
'IS_NUMBER_DEF'
   "(is_number (number n) = T) /\
    (is_number undef_num = F) ";;
IS_NUMBER_DEF = _
|- (!n. is_number(number n) = T) /\ (is_number undef_num = F)

let IS_UNDEFINED_DEF= new_recursive_definition false Number_Axiom
  'IS_UNDEFINED_DEF'
   "(is_undefined (number n) = F) /\
    (is_undefined undef_num = T)  ";;


#let GET_NUM_DEF = new_recursive_definition false Number_Axiom
    'GET_NUM_DEF'
   "(get_num (number n) = n) /\
    (get_num undef_num = 0) ";;

GET_NUM_DEF = |- (!n. get_num(number n) = n) /\ (get_num undef_num =
```

The store also needs to be lifted; its definition in HOL is as follows:

```
#let Store_Axiom = define_type 'Store'
  'Store = store (Identifier->Number) | undef_store';;
Store_Axiom =
|- !f e. ?! fn. (!f'. fn(store f') = f f') /\ (fn undef_store = e)
```

Once again, we have standard theorems for this type:

```
#let Store_Induct = prove_induction_thm Store_Axiom;;
Store_Induct =
|- !P. (!f'. P(store f')) /\ P undef_store ==> (!S'. P S')
#let Store_one_one =
  save_thm ('Store_one_one',
   prove_constructors_one_one Store_Axiom);;

Store_one_one = |- !f' f''. (store f' = store f'') = (f' = f'')
#let Store_distinct =
  save_thm ('Store_distinct',
   prove_constructors_distinct Store_Axiom);;
Store_distinct = |- !f'. ~(store f' = undef_store)
let Store_cases =
  save_thm ('Store_cases',
   prove_cases_thm Store_Induct);;
Store_cases = |- !S'. (?f'. S' = store f') \/ (S' = undef_store)
```

It is straightforward now to define the operations **newstore, access** and **update**. Note that the latter two (strict) functions are defined using **new_recursive_definition** to specify their action on every possible pattern.

```
#let NEWSTORE_DEF = new_definition ('NEWSTORE_DEF',
  "(newstore:Store) =  store (\i.undef_num)");;
#NEWSTORE_DEF = |- newstore = store(\i. undef_num)

#let ACCESS_DEF = new_recursive_definition false Store_Axiom
 'ACCESS_DEF'
  "(access i (store s)  = s i) /\
   (access i undef_store = undef_num)";;

ACCESS_DEF =
|- (!i s. access i(store s) = s i) /\
   (!i. access i undef_store = undef_num)

#let UPDATE_DEF = new_recursive_definition false Store_Axiom
 'UPDATE_DEF'
  "(update i v (store m)  = store (\j.(j=i) => v | m j)) /\
   (update i v undef_store = undef_store)";;

UPDATE_DEF =
|- (!i v m. update i v(store m) = store(\j. ((j = i) => v | m j))) /\
   (!i v. update i v undef_store = undef_store)
```

This completes the HOL formalisation of the semantic algebras for IMP1.

5.2.2 Syntax

The syntax for IMP1 is neatly captured using HOL's type package. Below we give the syntax for expressions, with the usual auxiliary theorems. The HOL responses are rather long, so they are omitted.

Note that we are using an abstract syntax in which, say, `Plus` e_1 e_2 is used to render the original e_1 + e_2 in the concrete syntax. This is necessary because the type package requires constructors to be of this form. It does have the advantage of giving the language phrases a uniform appearance. From one point of view, only the abstract syntax of the language matters; however, the big disadvantage is that programs written using it can quickly become unreadable, and are also difficult to write because of the morass of brackets. Clearly a parser (from concrete to abstract syntax) is needed so that programs can be easily written. This is easily done using the Unix tools yacc and lex.

For boolean expressions and commands we have, similarly:

```
let BExpression_Axiom = define_type 'BExpression'
   'BExpression =
               True |
               False |
               Equals Expression Expression |
               Not BExpression';;

let BExpression_Induct = prove_induction_thm BExpression_Axiom;;

save_thm ('BExpression_one_one',
   prove_constructors_one_one BExpression_Axiom);;

save_thm ('BExpression_distinct',
   prove_constructors_distinct BExpression_Axiom);;

save_thm ('BExpression_cases',
  prove_cases_thm BExpression_Induct);;
```

```
let Command_Axiom = define_type 'Command'
   'Command   =   Skip |
                  Val Identifier Expression |
                  If BExpression Command Command |
                  While BExpression Command |
                  Seq Command Command |
                  Diverge' ;;

let Command_Induct = prove_induction_thm Command_Axiom;;

save_thm ('Command_one_one',
   prove_constructors_one_one Command_Axiom);;

save_thm ('Command_distinct',
   prove_constructors_distinct Command_Axiom);;

save_thm ('Command_cases',
  prove_cases_thm Command_Induct);;
```

It will be convenient to allow a list of commands (this is a form of syntactic sugar):

```
let SEQL_DEF = new_list_rec_definition ('SEQL_DEF',
       "(Seql [] = Skip) /\
        (Seql (CONS command sequence) =
           Seq command  (Seql sequence))");;
SEQL_DEF =
 - (Seql[] = Skip) /\
    (!command sequence.
      Seql(CONS command sequence) = Seq command(Seql sequence)
```

To complete the syntactic description, we have

```
#new_type_abbrev ('Program', ":Command");;
```

5.2.3 Semantic Equations

Now we are ready to capture the semantics of IMP1 in HOL. To make things as simple as possible, we shall make use of the fixed-point combinator in order to express the semantics of while loops. The fixed-point property is added as an axiom to the theory (this is reminiscent of LCF, where fixed-points and partial orderings are part of the logic — not proved using some mathematical model). Clearly, this is a short-cut, and quite an unsound thing to do[3]. We must promise only to use appropriate functions f for which domain theory guarantees us that FIX f is well-defined!

```
#new_constant ('FIX', ":(*->*)->*");;

#let FIX_EQ = new_axiom ('FIX_EQ', "!f:*->*.FIX f = f (FIX f)");;
FIX_EQ = |- !f. FIX f = f(FIX f)
```

The semantic equations for expressions are given in terms of the valuation function EXPR, which is naturally defined using `new_recursive_definition`:

[3] It would be sounder to import results from a HOL theory of domains. Such a theory has been constructed by Albert Camilleri, but will not be used in this paper.

```
#let EXPR_DEF = new_recursive_definition false Expression_Axiom
   'EXPR_DEF'
   "(EXPR (Const v) s = ((s = undef_store) => undef_num | v)) /\
    (EXPR (Var i) s = access i s) /\
    (EXPR (Plus e1 e2) s =
      (is_number (EXPR e1 s) /\ is_number (EXPR e2 s))
 => number (get_num (EXPR e1 s) + get_num (EXPR e2 s)) | undef_num)";;

EXPR_DEF =
 - (!v s. EXPR(Const v)s = ((s = undef_store) => undef_num | v)) /\
   (!i s. EXPR(Var i)s = access i s) /\
   (!e1 e2 s.
     EXPR(Plus e1 e2)s =
     ((is_number(EXPR e1 s) /\ is_number(EXPR e2 s)) =>
      number((get_num(EXPR e1 s)) + (get_num(EXPR e2 s))) |
      undef_num))
```

For boolean expressions we have

```
#let BOOL_EXPR_DEF = new_recursive_definition false BExpression_Axiom
  'BOOL_EXPR_DEF'
  "(BOOL_EXPR True s = T) /\
   (BOOL_EXPR False s = F) /\
   (BOOL_EXPR (Equals e1 e2) s = (EXPR e1 s = EXPR e2 s))  /\
   (BOOL_EXPR (Not b) s = ~(BOOL_EXPR  b s))";;

BOOL_EXPR_DEF =
 - (!s. BOOL_EXPR True s = T) /\
   (!s. BOOL_EXPR False s = F) /\
   (!e1 e2 s. BOOL_EXPR(Equals e1 e2)s = (EXPR e1 s = EXPR e2 s)) /\
   (!b s. BOOL_EXPR(Not b)s = ~BOOL_EXPR b s)
```

The semantics of commands and programs is given by similar functions, after which we close the theory.

```
#let COMMAND_DEF = new_recursive_definition false Command_Axiom
   'COMMAND_DEF'
   "(COMMAND Skip s = s) /\
    (COMMAND (Val i e) s =
       (s = undef_store) => undef_store | update i (EXPR e s) s) /\
    (COMMAND (If b c1 c2) s =
       (s = undef_store) => undef_store |
        ((BOOL_EXPR b s) => (COMMAND c1 s) | (COMMAND c2 s)))  /\
   (COMMAND (While b c) s=
        FIX (\f t.(BOOL_EXPR b t => f (COMMAND c t) | t)) s) /\
  (COMMAND (Seq c1 c2) s =
       (s = undef_store) => undef_store | COMMAND c2 (COMMAND c1 s)) /\
    (COMMAND Diverge s = undef_store)";;
COMMAND_DEF =
 - (!s. COMMAND Skip s = s) /\
   (!i e s.
     COMMAND(Val i e)s =
     ((s = undef_store) => undef_store | update i(EXPR e s)s)) /\
   (!b c1 c2 s.
     COMMAND(If b c1 c2)s =
     ((s = undef_store) =>
      undef_store |
      (BOOL_EXPR b s => COMMAND c1 s | COMMAND c2 s))) /\
   (!b c s.
     COMMAND(While b c)s =
     FIX(\f t. (BOOL_EXPR b t => f(COMMAND c t) | t))s) /\
    !c1 c2 s.
     COMMAND(Seq c1 c2)s =
      (s = undef_store) => undef_store | COMMAND c2(COMMAND c1 s)) /\
    !s. COMMAND Diverge s = undef_store)
#let PROGRAM_DEF = new_definition ('PROGRAM_DEF',
   "PROGRAM (p:Program) = \n.let s = (update 'input' n newstore) in
      let s' = (COMMAND p s) in (access 'output' s')");;

PROGRAM_DEF =
|- !p.
    PROGRAM p =
    (\n.
      let s = update 'input' n newstore
      in
       let s' = COMMAND p s in access 'output' s')

#close_theory ();;
```

5.3 Reasoning about Programs in IMP1

5.3.1 Tactics

As with any user-constructed theory, we need a number of tactics which are appropriate for the kind of problems we wish to solve. In the following STRING_RULE recursively searches terms and rewrites string expressions such as 'a' = 'a' as T (true) and 'a' = 'b' to F (false); STRING_TAC is the corresponding tactic. Next we have rules and tactics for comparing and adding numbers, and for comparing functions. Finally, we shall need two tactics which carry out case analysis on variables of type Store and Command.

```
let STRING_TAC       =  CONV_TAC  (DEPTH_CONV string_EQ_CONV);;
let STRING_RULE      =  CONV_RULE (DEPTH_CONV string_EQ_CONV);;


let ADD_TAC          =  CONV_TAC (ONCE_DEPTH_CONV ADD_CONV);;
let ADD_RULE         =  CONV_RULE (ONCE_DEPTH_CONV ADD_CONV);;

let num_EQ_TAC       =  CONV_TAC (ONCE_DEPTH_CONV num_EQ_CONV);;
let num_EQ_RULE      =  CONV_RULE (ONCE_DEPTH_CONV num_EQ_CONV);;

let FUN_EQ_TAC       =  CONV_TAC (DEPTH_CONV FUN_EQ_CONV);;
let FUN_EQ_RULE      =  CONV_RULE (DEPTH_CONV FUN_EQ_CONV);;

let Command_CASES_TAC t = DISJ_CASES_THEN
                          STRIP_ASSUME_TAC (SPEC t Command_cases) THEN
                          (ASM_REWRITE_TAC [Command_distinct]);;

let Store_CASES_TAC t = DISJ_CASES_THEN
                          STRIP_ASSUME_TAC (SPEC t Store_cases) THEN
                          (ASM_REWRITE_TAC [Store_distinct]);;
```

Two auxiliary lemmas are needed (the proofs are omitted):

```
LEMMA1 = |- !s. ~(s = undef_store) => ~(update i v s = undef_store)
LEMMA2 = |- ~(update i v newstore = undef_store)
```

Next we define SIMPLIFY_TAC, which repeatedly uses the definitions of the semantic operations, along with some basic arithmetic, and simplifies environments using BETA_TAC and STRING_TAC. If these steps are making no progress, the goal is rewriten once, in case we need to unfold a term involving a fixed-point combinator.

```
let DEFS = [IS_UNDEFINED_DEF;
       IS_NUMBER_DEF;
       GET_NUM_DEF;
       ACCESS_DEF;
       UPDATE_DEF;
       NEWSTORE_DEF;
          Number_one_one;
          Number_distinct;
          Store_one_one;
          Store_distinct;
          ADD_CLAUSES;
          LEMMA2];;

let SIMPLIFY_TAC =
      REPEAT
         (CHANGED_TAC
          (ASM_REWRITE_TAC DEFS THEN
           BETA_TAC THEN
           STRING_TAC THEN
           num_EQ_TAC THEN
           ADD_TAC ORELSE
          (ONCE_REWRITE_TAC [FIX_EQ])));;
```

Our final collection of tactics basically carry out rewriting with the semantic equations.

```
let EXPR_TAC        =  REWRITE_TAC [EXPR_DEF] ;;

let BOOL_EXPR_TAC   =  REWRITE_TAC [BOOL_EXPR_DEF];;

let COMMAND_TAC     =  REWRITE_TAC [COMMAND_DEF; SEQL_DEF];;

let PROGRAM_TAC     =  REWRITE_TAC [LET_DEF; PROGRAM_DEF]
                       THEN BETA_TAC;;

let RUN_TAC         =  PROGRAM_TAC THEN
                       COMMAND_TAC THEN
                       BOOL_EXPR_TAC THEN
                       EXPR_TAC;;
```

The tactics we have defined may look rather *ad hoc* — and are to some extent — but are very simple in structure and easy to comprehend. They illustrate the fact that powerful new tactics are easy to construct (using tacticals).

5.3.2 Example Proofs

We shall give a number of example proofs of correctness about programs in IMP1 using the above tactics.

First, some basic constants:

```
let v0 = "number 0";;
let v1 = "number 1";;
let v2 = "number 2";;
let v3 = "number 3";;
let v4 = "number 4";;
let e0 = "Const ^v0";;
let e1 = "Const ^v1";;
let e2 = "Const ^v2";;
let e3 = "Const ^v3";;
let e4 = "Const ^v4";;
```

First we shall prove the example already given in Chapter 2

$$\boxed{\begin{array}{l} P[\mathbf{output} := 1;\ \mathbf{if\ input} = 0\ \mathbf{then\ diverge\ else\ skip};\ \mathbf{output} := 3] = \\ \qquad \lambda n.n = 0 \Rightarrow \bot \,\Box\, 3 \end{array}}$$

The method of proof is simple: we rewrite successively with the semantic equations, and then perform case analysis on n = 0. Finally, we simplify.

```
#let p = "Seql [Val 'output' ^e1;
                   If (Equals (Var 'input') ^e0)
                        Diverge
                   % else %
                         Skip;
                    Val 'output' ^e3]";;

g "PROGRAM ^p n = (n = ^v0) => undef_num | ^v3";;
"PROGRAM
  (Seql
   [Val 'output'(Const(number 1));
    If(Equals(Var 'input')(Const(number 0)))Diverge Skip;
    Val 'output'(Const(number 3))])
  n =
   (n = number 0) => undef_num | number 3)"

e (PROGRAM_TAC THEN
   COMMAND_TAC THEN
   BOOL_EXPR_TAC THEN
   EXPR_TAC THEN
   ASM_CASES_TAC "n = ^v0" THEN
   SIMPLIFY_TAC);;
OK..
goal proved
|- PROGRAM
    (Seql
     [Val 'output'(Const(number 1));
      If(Equals(Var 'input')(Const(number 0)))Diverge Skip;
      Val 'output'(Const(number 3))])
    n =
    ((n = number 0) => undef_num | number 3)

Previous subproof:
goal proved
```

This is quite a simple program, but it immediately highlights some problems. Firstly, it is very slow (on a Sun Sparcstation 1+, this proof took about 7 minutes, and generated more than 46,000 primitive inferences). Secondly, if the proof is done interactively, the intermediate goals are horribly long (taking five or more screens to display). This is because there is as yet no straightforward facility for abbreviating terms in HOL.

Our second example shows the unfolding of a `while` loop (the proof takes about 15 minutes):

$$P[\mathbf{output} = 0;\ \mathbf{while}(\mathbf{not}(\mathbf{input} = 1))\ \mathbf{input} = \mathbf{input} + 1;\ \mathbf{output} = \mathbf{output} + 1;]\ 0 = 1$$

```
g "PROGRAM
   (Seql [Val 'output' ^e0;
          While (Not (Equals (Var 'input') ^e1))
                (Seql [Val 'input' (Plus (Var 'input') ^e1);
                       Val 'output' (Plus (Var 'output') ^e1)])])
                  ^v0 = ^v1";;
e (RUN_TAC THEN
   SIMPLIFY_TAC THEN
   RUN_TAC THEN
   SIMPLIFY_TAC);;
OK..
goal proved
|- PROGRAM
    (Seql
     [Val 'output'(Const(number 0));
      While
      (Not(Equals(Var 'input')(Const(number 1))))
      (Seql
       [Val 'input'(Plus(Var 'input')(Const(number 1)));
        Val 'output'(Plus(Var 'output')(Const(number 1)))])])
    (number 0) =
    number 1

Previous subproof:
goal proved
```

Several examples of equivalence have also been proved using HOL, including the following:

$$e_1 + e_2 \approx e_2 + e_1$$

$$c;\ \text{skip} \approx c$$

$$c;\ \text{diverge} \approx \text{diverge}$$

We shall give the proofs of just two equivalences:

$$\textbf{if b then } c_1 \textbf{ else } c_2 \approx \textbf{if(not b) then } c_2 \textbf{ else } c_1$$

```
g "COMMAND (If b c1 c2) = COMMAND (If (Not b) c2 c1)";;

e (FUN_EQ_TAC THEN
   GEN_TAC THEN
   COMMAND_TAC THEN BOOL_EXPR_TAC THEN
   COND_CASES_TAC THEN COND_CASES_TAC THENL
      [SIMPLIFY_TAC ;
       SIMPLIFY_TAC;
       TRIVIAL_TAC ;
       REWRITE_TAC []]);;
OK..
goal proved
|- COMMAND(If b c1 c2) = COMMAND(If(Not b)c2 c1)

Previous subproof:
goal proved
```

$$\mathbf{X := 0;\ Y := X + 1 \approx Y := 1;\ X := 0}$$

```
g "COMMAND (Seql [Val 'x' ^e0;
                  Val 'y' (Plus (Var 'x') ^e1)]) =
   COMMAND (Seql [Val 'y' ^e1; Val 'x' ^e0])";;
e (FUN_EQ_TAC THEN
   GEN_TAC THEN
   COMMAND_TAC THEN
   EXPF_TAC THEN
   Store_CASES_TAC  "S':Store" THEN
   SIMPLIFY_TAC THEN
   FUN_EQ_TAC THEN
   BETA_TAC THEN
   GEN_TAC THEN
   (COND_CASES_TAC THEN COND_CASES_TAC) THEN
   SIMPLIFY_TAC THEN
   (REWRITE_ASL_WITH_ASL_TAC []) THEN
   (CHANGE_ASM_TAC 1 STRING_RULE) THEN
   ASM_REWRITE_TAC []);;
goal proved
|- COMMAND
   (Seql
    [Val 'x'(Const(number 0));
     Val 'y'(Plus(Var 'x')(Const(number 1)))]) =
   COMMAND(Seql[Val 'y'(Const(number 1));Val 'x'(Const(number 0))])

Previous subproof:
goal proved
```

Chapter 6 ISABELLE

In this chapter, we shall discuss the theorem prover Isabelle, which has been under development by Larry Paulson at the University of Cambridge since 1986. Isabelle is also a descendant of the LCF system. However, it is based on quite different basic concepts, and has a number of features which make it different from HOL. As in HOL, formulae are manipulated by the language ML (in this case Standard ML), and the system provides for backwards proof by means of tactics and tacticals.

The main source for this chapter is the Isabelle manual [25]. The theory underlying Isabelle is discussed in [26, 27].

6.1 Basic Concepts

Isabelle is a generic theorem prover: the logic of discourse is not fixed, but can be chosen from a number of built-in logics provided with the system, or even defined *ab initio* (although this is very difficult). Isabelle has an expressive *meta-logic*, in which the inference rules and axioms of *object logics* can be formulated. Isabelle comes with a number of object logics, including First Order Logic (FOL), Higher Order Logic (HOL) and Constructive Type Theory (CTT).

Note: when n .cessary to avoid confusion we use "HOL" to refer to Mike Gordon's HOL system, and Isabelle-HOL to refer to Higher Order Logic as captured as an object logic within Isabelle.

6.1.1 Isabelle's Meta-Logic

The meta-logic used in Isabelle is intuitionistic higher order logic with universal quantification and equality. It was chosen as being the minimal logic capable of formulating the axioms and rules of arbitrary object logics.

The table below shows the constructs used in the meta-logic, and their keyboard equivalents

Table 8 Isabelle Meta-Logic Constructs

Notation	Keyboard	Description
$a = b$	a == b	meta-equality
$\phi \Rightarrow \psi$	ϕ ==> ψ	meta-implication
$\phi_1 \Rightarrow (\ldots\phi_n \Rightarrow \psi)$	[| $\phi_1, \ldots \phi_n$ |] ==> ψ	nested implication
$\Lambda x.\phi$	!x.ϕ	meta-quantification
$\lambda x.\phi$	%x.ϕ	meta-abstraction
?P	?P	scheme variables

Pure Isabelle contains the material common to all logics: theories, rules, tactics, subgoal commands, types and terms.

6.1.2 Object Logics

An object logic is an ML object of type **theory**. The axioms and rules are of type **thm**. Various symbols are used in object logics; some of their keyboard equivalents are given below:

Table 9 Object Logic Symbols

Notation	Keyboard	Description
$P \supset Q$	P --> Q	Implication
$P \leftrightarrow Q$	P <-> Q	Bi-implication
$\forall x.P$	ALL x.P	Universal quantification
$\exists x.P$	EX x.P	Existential quantification
$\neg P$	~ P	Negation

Isabelle emphasises the natural style of reasoning. To illustrate this, we give the natural deduction system for intuitionistic first order logic. Each logical connective may have elimination or introduction rules. For example the rule for implication elimination (denoted by ⊃ E) is just the well-known law of *modus ponens* in first-order logic.

Table 10 Intuitionistic First Order Logic

	Introduction (I)	Elimination (E)
Conjunction	$\dfrac{A \quad B}{A \& B}$	$\dfrac{A \& B}{A}$ $\dfrac{A \& B}{B}$
Disjunction	$\dfrac{A}{A \vee B}$ $\dfrac{B}{A \vee B}$	$\dfrac{A \vee B \quad [A]\, C \quad [B]\, C}{C}$
Implication	$\dfrac{[A] \quad B}{A \supset B}$	$\dfrac{A \supset B \quad A}{B}$
Contradiction		$\dfrac{\bot}{A}$

Table 10 (Continued) Intuitionistic First Order Logic

Universal Quantifier	$\frac{A}{\forall x.A}$	$\frac{\forall x.A}{A[t/x]}$
Existential Quantifier	$\frac{A[t/x]}{\exists x.A}$	$\frac{\exists x.A \quad [A]\ B}{B}$

6.1.3 Inference Rules

Isabelle works with inference rules expressed in a natural deduction style. We shall show how inference rules are "packaged" within Isabelle. Consider the rule &I. In the meta-logic, this is expressed as follows:

$$\bigwedge AB.A \Rightarrow (B \Rightarrow A\&B)$$

which eventually is rendered into the following keyboard characters:

```
[|A; B|] ==> A&B
```

(Actually, the variables involved are treated internally as scheme variables ?A and ?B, which may be instantiated during unification).

6.1.4 Subgoal Package

As with HOL, Isabelle carries out goal-directed proofs, and contains a subgoal package to assist with interactive proof. A *proof state* consists of a *goal*, along with a number of subgoals whose validity establishes that of the goal. The subgoals can be thought of as proof obligations. Diagrammatically we display a proof state as follows:

$$\begin{array}{c}\text{goal}\\ \hline \text{subgoal}_1\\ \text{subgoal}_2\\ \cdots\end{array}$$

When we set a goal in Isabelle we have as our initial proof state

$$\begin{array}{c}\text{goal}\\ \hline \text{goal}\end{array}$$

in which there is a single subgoal identical with the original goal. When we reach a proof state with no subgoals, we clearly have a proof of the original goal. As in HOL, *tactics* are available to transform proof states to new proof states, using the ML function by. However, there is a crucial difference. In LCF and HOL, a tactic either gives a unique new proof state, or fails. But in Isabelle a tactic can return more than one, and possibly even an infinite number, of new proof states (how this can happen we shall see shortly). If T is a tactic, and ϕ a proof state, then the result $T\phi$ of applying T to ϕ is written as a list to capture the various alternatives:

$$\begin{array}{ll} T\phi = [\ ] & \text{(failure)}\\ T\phi = [\psi] & \text{(unique result)}\\ T\phi = [\psi_1, \psi_2, \psi_3, \ldots] & \text{(multiple outcomes)}\end{array}$$

The possibility that a tactic can have multiple outcomes has a profound effect on the way one thinks about theorem proving in Isabelle. The user tends to think on a grander scale: a proof state is usually presented with all of its subgoals shown (contrast this with HOL, where one only sees one subgoal at a time), and a number of proof strategies act on a number of subgoals, automatically instantiating variables and renumbering the subgoals as appropriate.

6.1.5 Tactics

Pure Isabelle has a number of basic tactics (object logics come with a number of special purpose tactics). We shall discuss the most important of these.

Recall that Isabelle emphasises the natural style of reasoning; correspondingly, most proof steps are carried out backwards reasoning using inference rules of the theory. This is called *resolution*. Isabelle provides a single ML function to do this (again, this is an improvement over HOL, where a new tactic written in ML must be provided for each new inference rule).

The basic resolution tactic is `resolve_tac thms i`. This tactic tries each theorem (object logic rule) in the list `thms` against subgoal i of the proof state. For a given rule, say

$$[| B_1, \ldots, B_k |] ==> B$$

resolution can form the next state by unifying the conclusion with the subgoal, replacing it by the instantiated premises. Thus if the subgoal is

$$[| A_1, \ldots, A_n |] ==> A$$

and A can unify with B, resolution will produce the following new subgoals:

$$[| \overline{A_1}, \ldots, \overline{A_n} |] ==> \overline{B_1}$$

$$\ldots$$

$$[| \overline{A_1}, \ldots, \overline{A_n} |] ==> \overline{B_k}$$

in which the overbars denote the resulting formulae after instantiations have been made. Subgoals frequently change their appearance as instantiations propagate throughout the proof tree — something which users of HOL will find strange at first.

Note that unification in Isabelle is full higher-order unification (ie solving equations in the typed λ-calculus with respect to α, β and η conversion). There can be multiple outcomes, arising from the fact that there can be more than one higher-order unifier. Multiple outcomes can also arise if more than one theorem can be resolved with the goal. The tactic will fail if none of the rules can be unified.

Another fundamental tactic is `assume_tac i`, which tries to solve subgoal i by assumption (again, this may involve unification).

Reasoning about definitions and deriving new rules is facilitated by a number of rewriting tactics. For example, `rewrite_goals_tac thms` uses the given definitional theorems for rewriting subgoals. Rewriting is not as prominent in Isabelle as it is in HOL. In fact, it is frowned upon: if we define a new construct in a theory, the preferred strategy is to derive immediately elimination and introduction inference rules for the construct, and thereafter to use these new rules in resolution steps. Rewriting with the original definition can introduce unwanted complexity to a proof.

6.1.6 A Simple Proof

To illustrate theorem proving in Isabelle, consider a simple proof, namely the obvious fact in first-order logic that:

$$P\&Q \supset (R \supset P\&R)$$

A step-by-step proof is given below. We first set the goal, and then resolve twice with the rule $\supset$I. The conjunction is then attacked by resolving with &I. This gives two subgoals, the second of which is solved by assumption, and the first by &E1. Using assumption again solves the goal: the result is a theorem which Isabelle echoes with scheme variables.

```
- goal Int_Rule.thy "P&Q --> (R --> P&R)";
Level 0
P & Q --> R --> P & R
 1. P & Q --> R --> P & R;

- by (resolve_tac [imp_intr] 1);
Level 1
P & Q --> R --> P & R
 1. P & Q ==> R --> P & R


- by (resolve_tac [imp_intr] 1);
Level 2
P & Q --> R --> P & R
 1. [| P & Q; R |] ==> P & R

- by (resolve_tac [conj_intr] 1);
Level 3
P & Q --> R --> P & R
 1. [| P & Q; R |] ==> P
 2. [| P & Q; R |] ==> R

- by (assume_tac 2);
Level 4
P & Q --> R --> P & R
 1. [| P & Q; R |] ==> P

- by (resolve_tac [conjunct1] 1);
Level 5
P & Q --> R --> P & R
 1. [| P & Q; R |] ==> P & ?Q3

- by (assume_tac 1);
Level 6
P & Q --> R --> P & R
No subgoals!

prth (result());
?P & ?Q --> ?R --> ?P & ?R
```

6.1.7 Tacticals

Single-step proofs such as the one above are much too laborious. As with LCF and HOL, *tacticals* are available to build new tactics from basic tactics. A selection of basic tacticals is as follows:

tac1 THEN tac2
tac1 ORELSE tac2
REPEAT tac
DEPTHFIRST pred tac

However, because tactics can have multiple outcomes, these tacticals are more high-powered than their counterparts in LCF and HOL. They work by combining sequences of proof states.

The tactic `tac1 THEN tac2`, applied to the proof state ϕ, first computes `tac1(`ϕ`)`, giving some list [ψ_1, ψ_2, ...] of proof states, and then applies `tac2` to each of these states, giving as output the concatenation of the sequences `tac2(`ψ_1`)`, `tac2(`ψ_2`)`, ...

The tactic tac1 ORELSE tac2 is a form of choice: it first computes tac1 (ϕ). If this is non-empty, it is returned as the result; otherwise, tac2 (ϕ) is returned.

The tactic REPEAT tac first computes tac (ϕ). If this is non-empty, then the tactics recursively applies itself to each element, concatenating the results. Otherwise, it returns [ϕ].

The tactic DEPTH_FIRST pred tac performs a depth-first search for a proof-state satisfying pred. Usually pred is taken to be "no subgoals", so that the tactic will search for a proof of the original goal.

To show the power of tacticals, we do our example proof again using a single tactic:

$$P\&Q \supset (R \supset P\&R)$$

```
- goal Int_Rule.thy "P&Q --> (R --> P&R)";
Level 0
P & Q --> R --> P & R
 1. P & Q --> R --> P & R;

- by (REPEAT
      (assume_tac 1 ORELSE
       resolve_tac [imp_intr, conj_intr, conjunct1] 1));

Level 1
P & Q --> R --> P & R
No subgoals!
```

Even more dramatic is the built-in tactic fast_tac which is powerful enough to solve a large class of basic goals in logic:

```
- goal Int_Rule.thy "P&Q --> (R --> P&R)";
Level 0
P & Q --> R --> P & R
 1. P & Q --> R --> P & R;

- by (fast_tac [] 1);

Level 1
P & Q --> R --> P & R
No subgoals!
```

6.1.8 Comments

We omit here a discussion of how new object logics are constructed in Isabelle: this is by far the most difficult aspect of the system. In the next chapter, we shall show how simple extension of an existing theory (such as Isabelle-HOL) can provide us with a program verification environment.

Chapter 7 PROGRAM VERIFICATION IN ISABELLE

7.1 Introduction

In this chapter we shall discuss some experiments in reasoning about programs using Isabelle.

We showed in Chapter 5 that HOL can be used to reason about denotational semantics definitions. In principle, we could use Isabelle-HOL (i.e., Isabelle's object logic HOL) in the same way: definitions can be set up, and proof steps devised exactly as was done for the HOL system. Rewriting would take care of a lot of the proof steps. The advantage of using Isabelle in this way would be the possibility of answer extraction (using scheme variables). The disadvantages are that Isabelle is better at using derived inference rules than rewriting with definitions, and that the built-in theories provided with Isabelle are not as extensive as those in the HOL system.

We also claimed in Chapter 5 that HOL was not as well-suited to the study of operational semantics. In contrast, Isabelle looks ideal for this purpose, because it stresses the natural deduction style, and works well with derived inference rules. If we regard our operational semantics as the rules for a logic, then we can quickly construct powerful proof procedures in Isabelle which allow reasoning about quite complicated programs.

For the above reasons, we shall concentrate on the implementation of operational semantics in Isabelle, using as working examples the languages FUNC1, FUNC2 and FUNC3 presented in Chapter 2. Our aim is to automate the following:

1. Evaluation of phrases, for example

$$\mathrm{E} \vdash \mathrm{let\ x = 7\ in\ x - 3} \Rightarrow ?\mathrm{n}$$

2. Proofs of correctness, for example: the swap routine

$$\{(\mathrm{x}, \mathrm{a}), (\mathrm{y}, \mathrm{a})\} \vdash \mathrm{let\ z = x;\ x = y;\ y = z\ in\ x} \Rightarrow \mathrm{b}$$

3. Reasoning about equivalence of program phrases, for example

$$\mathrm{let\ x = 1\ in\ x + 2} \approx \mathrm{let\ y = 2\ in\ y + 1}$$

7.2 The Language FUNC1

How can we implement the operational semantics for FUNC1? We have remarked before that it is difficult to set up new object logics in Isabelle from scratch. The easiest way to proceed is to extend an existing object logic. For our purposes, we shall extend the theory **arith** (which is an extension of **HOL**) to a new theory called **fl_thy**. It is important to note that we make no use of the features of HOL, except for the fact that it is a typed logic. The semantic domains such as identifiers and values, and the various kinds of language phrase (here expressions and declarations), are then regarded as HOL types. We could equally well have pictured them as sets, but this makes things more complicated.

We shall not discuss the Isabelle code in detail, but concentrate on the essentials (as an example, the complete source code for the larger language FUNC3 is given in the Appendix).

7.2.1 Syntax and Semantics

The first thing to be done is to capture the syntax of the language. We do this by means of a number of inference rules:

```
val f1_thy  =
  extend_theory arith_thy "f1" ...
```

```
[ ("Const_type",
     "n : nat ==> Const (n) : expr"),
  ("Var_type",
     "x : ide ==> Var (x) : expr"),
  ("Plus_type",
     "[| e1 : expr; e2 : expr |] ==>
     Plus (e1,e2) : expr"),
  ("Let_type",
     "[| d : decl; e : expr |] ==>
     Let (d,e) : expr"),
  ("Val_type",
     "[| x : ide; e : expr |] ==>
     Val (x,e) : decl"),
  ("Comp_type",
     "[| d1 : decl; d2 : decl |] ==>
     Comp (d1, d2) : decl"),
  ("empty_type",
     "empty : env"),
  ("bind_type",
     "[| E : env; x : ide; v : value |] ==>
     bind (x, v, E) : env")
  ("lookup1_rule",
     "[| E : env; x : ide;  n : nat |] ==>
       lookup (x, bind (x, n, E), n)" ),
  ("lookup2_rule",
     "[| E : env; x : ide; y : ide; v : value ;
       w : value;  ~(y = x : ide); lookup (x, E, n) |]
       ==> lookup (x, bind (y, w, E), n)" )
  ("combine_empty",
     "E : env ==> combine (E, empty, E)"),
  ("combine_bind",
     "[| E1 : env; E2 : env; E3 : env; x : ide; a : value;\
     \   combine (E1,E2,E3) |] ==>  \
     \   combine (E1, bind(x,a,E2), bind (x,a,E3))"),
...
```

These rules are very simple. For example, `Plus_type` expresses the fact that if e_1 and e_2 are expressions, then `Plus(`e_1`, `e_2`)` is also an expression. The constructor `Plus` corresponds to the symbol "+" in our language. Similar rules will hold for every other program phrase.

Environments need a little thought. We could implement an environment as any one of the following:

1. as a primitive type, augmented by axioms for a constructor function;
2. as a lambda-expression in HOL; or
3. as a list of pairs of type (expr,value)

The first approach was chosen because it reduces the dependency on the host object logic HOL to a minimum, and because it makes us assert as axioms exactly those inference rules needed to reason about environments. These rules are the last six given above. They assert the existence of a special environment called `empty` and a constructor function called `bind`. The lookup rules tell us how to look up values in the environment. The last two rules tell us all we need to know about combining environments, where combine(E,E',E") is a predicate which holds if E"=EE'.

The next step is to write down the rules of the operational semantics. In writing these rules, we render $E \vdash e \Rightarrow v$ as seq(E,e,v), and $E \vdash d \Rightarrow E'$ as seq'(E,d,E'). Here seq is read as "sequent". We give just the first three rules:

```
("Const_rule",
   "[| E : env; n : nat |] ==>
  seq (E, Const (n), n)" ),
"Var_rule",
  "[| E : env; x : ide; n : nat ;
 lookup (x, E, n) |]  ==> seq (E, Var (x), n)"),
"Let_rule",
  "[| E : env; E' : env; E'' : env; d : decl;
  e : expr; seq' (E, d, E');
  combine (E, E', E''); seq (E'', e, n) |] ==>
  seq (E, Let (d,e), n)"),
   ...
```

Once this has been done, we have the following lists of inference rules summarising the syntax and semantics of the language:

Table 11 Inference Rules: Syntax and Semantics

Name	Description
type_rules	inference rules capturing the syntax
lookup_rules	inference rules for bind
combine_rules	inference rules for combine
expr_rules	inference rules for expressions
decl_rules	inference rules for declarations
lang_rules	expr_rules @ decl_rules
common_asms	trivial assumptions taken for granted, such as: x : ide, y : ide m : nat, n : nat e : expr E : env

7.2.2 Proof Procedures

The construction of tactics to reason about programs in FUNC1 is quite straightforward. The basic strategy is to keep resolving on sequents (and attempting to reduce complex environments) until none remain, and then to apply the trivial type rules and common assumptions to finish off the proof. The only thing we need to be careful about is not to resolve any sequent of the form `seq (E, ?e, v)`, in which there is a scheme variable holding the place of an expression. Let us call such a sequent *unsafe*. If we were to resolve this sequent using the language rules, we would get multiple outcomes, and the proof effort will be wasted following wrong leads. Our method of getting over this is to define

$$\text{FIRSTONLY} : (\text{term}- > \text{bool})- > (\text{int}- > \text{tactic})- > \text{tactic}$$

to be a function which chooses the first subgoal for which a given selector function is true, and applies the given tactic, failing otherwise. Using this function, along with the selector function `is_safe_sequent`, which is true when the sequent is safe and false if not, we can define `step_tac`, which is the primitive proof step to remove language sequents. Our all-purpose tactic is fl_tac.

```
val reduce_tac =
    lookup_tac ORELSE combine_tac;

val step_tac   =
    FIRSTONLY is_safe_sequent (resolve_tac lang_rules);
val lang_tac   =
    REPEAT1 (step_tac THEN TRY (REPEAT1 reduce_tac));
```

```
val simp_tac   =
    REPEAT (ares_tac simp_rules 1);

val f1_tac     = lang_tac THEN simp_tac;
```

As an example, consider the proof of:

$$E \vdash \text{let } y = m \text{ in let } x = y \text{ in let } y = n \text{ in } x \Rightarrow m$$

We set the goal with a scheme variable in place of the answer, and use our tactic f1_tac:

```
goal f1_thy
   "seq (E, Let(Val(y,Const(const(m))),
              Let(Val(x,Var(y)),
          Let(Val(y,Const(const(n))),Var(x)))), ?a)";
by f1_tac;

Level 1
seq(E,Let(Val(y,Const(const(m))),
          Let(Val(x,Var(y)),
      Let(Val(y,Const(const(n))),Var(x)))),
    val(const(m)))
No subgoals!
```

Notice that Isabelle has obligingly given us the result of the evaluation, as well as proving the goal!

Our second example is the swap routine using a temporary variable:

$$\{(x, a), (y, b)\} \vdash \text{let } z = x;\ x = y;\ y = z \text{ in } x \Rightarrow b$$

```
goal f1_thy
   "seq (bind (x,a,bind(y,b,empty)),
                Let (Comp (Val (z, Var(x)),
                     Comp(Val(x, Var(y)),
        Val(y, Var(z)))), Var(x)), ?a)";
by f1_tac;

Level 1
seq(bind(x,a,bind(y,b,empty)),
    Let(Comp(Val(z,Var(x)),Comp(Val(x,Var(y)),
   Val(y,Var(z)))),Var(x)),b)
No subgoals!
```

7.3 The Language FUNC2

Now we shall consider how the semantics of FUNC2 may be implemented in Isabelle. The main hurdle to overcome is that a program phrase can have more than one rule associated with it. For example, recall from Chapter 2 that there are two rules for conditional expressions:

$$\frac{E \vdash e_0 \Rightarrow \mathbf{true} \qquad E \vdash e_1 \Rightarrow v}{E \vdash \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \Rightarrow v}$$

$$\frac{E \vdash e_0 \Rightarrow \mathbf{false} \qquad E \vdash e_2 \Rightarrow v}{E \vdash \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \Rightarrow v}$$

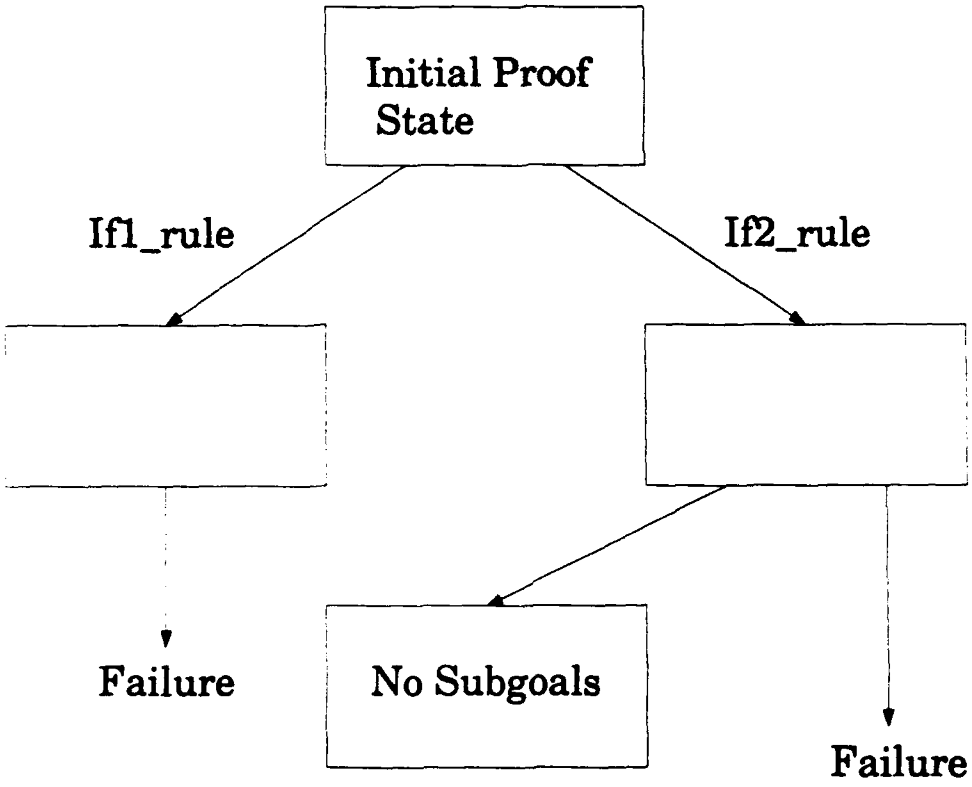
(which will be called If1_rule and If2_rule). Consider the following example:

$$\{(x, a), (y, b)\} \vdash \mathbf{if\ false\ then}\ x\ \mathbf{else}\ y \Rightarrow b$$

Suppose we try to prove this starting with **step_tac**:

```
- goal f2_thy
 "seq (bind (x, a, bind (y, b, empty)),
   If(Const(ff),Var(x),Var(y)), b)";
Level 0
seq(bind(x,a,bind(y,b,empty)),If(Const(ff),Var(x),Var(y)),b)
 1. seq(bind(x,a,bind(y,b,empty)),If(Const(ff),Var(x),Var(y)),b)
val it = [] : thm list
- by step_tac;
Level 1
seq(bind(x,a,bind(y,b,empty)),If(Const(ff),Var(x),Var(y)),b)
 1. bind(x,a,bind(y,b,empty)) : env
 2. Const(ff) : expr
 3. Var(x) : expr
 4. Var(y) : expr
 5. seq(bind(x,a,bind(y,b,empty)),Const(ff),val(tt))
 6. seq(bind(x,a,bind(y,b,empty)),Var(x),b)
val it = () : unit
```

Notice that the fifth subgoal is asserting that false evaluates to true — step_tac has chosen the wrong rule (namely If1_rule) to resolve on! We have gone down a "blind alley" in the proof.



To solve this problem, it will not be enough to rearrange the order of the language rules used by the resolution tactic, because there will always be examples for which the wrong rule is applied. What we need to do is use backtracking search. The appropriate tactics are

```
val step_tac  = FIRSTONLY is_safe_sequent (resolve_tac lang_rules);
val lang_tac  = REPEAT1 (step_tac THEN TRY (REPEAT1 reduce_tac)) THEN
          NOT_APPLICABLE is_sequent;

val simp_tac = REPEAT (ares_tac simp_rules 1);
val f2_tac    = DEPTH_FIRST (has_fewer_prems 1)
                           (lang_tac THEN simp_tac);
```

The tactic `f2_tac` carries out a depth-first backtracking search, looking for a proof state with no subgoals. We have modified `lang_tac` so that it will fail if the repeated application of `step_tac` and `reduce_tac` leaves behind any sequents. If we do not do this, `simp_tac` may give rise to infinite outcomes, and the search will not terminate.

```
- goal f2_thy
 "seq (bind (x, a, bind (y, b, empty)),
     If(Const(ff),Var(x),Var(y)), b)";
Level 0
seq(bind(x,a,bind(y,b,empty)),If(Const(ff),Var(x),Var(y)),b)
 1. seq(bind(x,a,bind(y,b,empty)),If(Const(ff),Var(x),Var(y)),b)
val it = [] : thm list
- by f2_tac;
Level 1
seq(bind(x,a,bind(y,b,empty)),If(Const(ff),Var(x),Var(y)),b)
No subgoals!
val it = () : unit
```

Our tactic is able to cope with the following well-known LISP example, in which x is used both as a bound and free variable:

$$\{(x,n)\} \vdash \text{let } (\text{val } z = \lambda f.\lambda x.f(f(x)));$$
$$\text{val } g = \lambda f.(f, x\ ) \text{ in } (z(g)\ m) \Rightarrow (m, n), n$$

```
- goal f2_thy
  "seq (bind (x, a, empty), \
  \  Let (Comp (Val (z,    \
  \        Fn (f, Fn (x, Apply (Var(f), Apply(Var(f),Var(x))))), \
             Val (g, Fn (f, Pr (Var(f), Var(x))))),     \
  \   Apply (Apply(Var(z), Var(g)), Const(const(m)))),     ?v)";
by (f2_tac);

...

Level 1
seq(bind(x,a,empty),
    Let(Comp(Val(z,Fn(f,Fn(x,Apply(Var(f),Apply(Var(f),Var(x)))))),
             Val(g,Fn(f,Pr(Var(f),Var(x))))),
        Apply(Apply(Var(z),Var(g)),Const(const(m)))),
    pair(pair(val(const(m)),a),a))
No subgoals!
val it = () : unit
```

The proof took only 75 seconds, including a garbage collection, which compares extremely well with the HOL system.

7.4 The Language FUNC3

The extension to FUNC3 is not difficult. We need to give function closures a fourth argument, as described in Chapter 2. We also need to add inference rules for the unfolding operation on environments. To control the search, it also becomes necessary to make tactics such as `lookup_tac`, `combine_tac` and `unfold_tac` safer, in that they are forced to fail if the environment being examined contains too many scheme variables.

Isabelle correctly evaluates the resulting environment for the standard definition of the factorial function:

$$\mathrm{E} \vdash \mathrm{rec}(\mathrm{val}\ \mathrm{f} = \lambda \mathrm{x}.\mathrm{if}(\mathrm{x} = 0)\ \mathrm{then}\ \mathrm{x} * \mathrm{f}(\mathrm{x} - 1))$$

Work on this language is continuing, and we shall not give any other detailed examples here.

7.5 The Language IMP1

Implementing operational semantics for imperative languages is just as straightforward. Earlier, we described part of the operational semantics for the language IMP1. The language has been fully implemented in Isabelle (with environments present as place-holders to allow for the eventual addition of local scoping) and some non-trivial proofs carried out. A translator has also been built to allow programs to be written in a simple syntax.

For example, consider the following program containing a while loop:

$$\mathrm{E} \vdash \mathrm{x} := 0;\ \ \mathrm{while}\ (\mathrm{x} <= 0)\ \mathrm{x} := \mathrm{x} + 1, \mathrm{M} \Rightarrow ?\mathrm{M}$$

Isabelle proves this rather quickly, and extracts the answer (the resulting memory):

```
- goal il_thy
"seq'(E, Comp (Assign (x, Const (0)),
      While (Less_equals (Var (x), Const (0)),
      Assign (x, Plus (Var (x), Const (Succ(0)))))), M, ?M) ";

seq'(E,Comp(Assign(x,Const(0)),
            While(Less_equals(Var(x),Const(0)),
                  Assign(x,Plus(Var(x),Const(Succ(0)))))),M,
     update(x,constant(0 #+ Succ(0)),update(x,constant(0),M)))
No subgoals!
val it = () : unit
```

Some arithmetic is required to proof programs such as these correct, and Isabelle has at the moment only limited support for arithmetic. Despite this, I believe that Isabelle is a powerful and natural tool for the study of operational semantics.

Chapter 8 DISCUSSION AND CONCLUSIONS

Our work has highlighted some of the advantages and disadvantages of using the automated theorem provers HOL and Isabelle for the study of language semantics and program verification. We begin with some specific comments on the HOL system.

8.1 Comments on HOL

8.1.1 Ease of Use

HOL is certainly a difficult system for a beginner to learn. The large number of theorems available, as well as the fine-grained nature of most of the inference rules and tactics, means that the new user must invest considerable time and effort in before even simple proofs can be attempted. The first taste of HOL can be quite frustrating, especially without a HOL "expert" to be a guide. There are a number of program verification/theorem proving environments — such as mEVES , Gypsy and MALPAS — which are simpler, and whose proof commands may even be more powerful. However, these systems are not flexible. Once the user gains familiarity with HOL, its expressive power and flexibility becomes more and more apparent. I believe that HOL is an invaluable tool —not just as a proof assistant, but as an aid to the clean formulation of mathematical theories.

8.1.2 Expressiveness

Higher order logic is a large and powerful logic; because it is higher order and polymorphic, it is expressive enough to be able to capture the most sophisticated mathematical theories. It allows denotational semantics definitions to be expressed in a natural and succinct way. Also, the soundness and level of mathematical rigour of the HOL system give us a high level of confidence in the proofs which it generates.

Reasoning about programs requires some fairly subtle mathematical techniques. The built-in inference rules and tactics of HOL are of a fine-grained nature, providing the user with a flexibility not available in many other automated reasoning systems. If a theorem or goal does not exactly fit the form required by a built-in tactic or inference rule, it is possible to carry out quite delicate manipulation until the required form is achieved.

8.1.3 Documentation

The new documentation which has appeared with the release of HOL88 makes it possible for a user working alone to make a lot of headway in understanding the system, and how to construct proofs. The manuals [22] are well-written and informative; clearly much care has gone into their production. The document called DESCRIPTION is a good overview. The TUTORIAL has excellent sections on parity checking, protocol verification and modular arithmetic, but I think could benefit from having many more short examples. I believe that the case study by Jeff Joyce on Microprocessor Systems is a fine piece of work, but seems out of place in a tutorial. There is rather too much emphasis on hardware verification (though this is natural, given the interest of the Cambridge group in this area). The REFERENCE manual is still incomplete, but will be greatly improved in the near future. It would be useful to have the tactics listed by type (as well as alphabetically).

8.1.4 Tactics

The HOL system's expressiveness is especially apparent in the way that new tactics can be programmed, either in the meta-language ML or by the use of tacticals. These tactics can be specially tailored for the problem domain being studied. Our aim in verifying programs was to derive tactics which capture proof procedures as general as possible, without sacrificing efficiency or clarity.

There are by now a large number of useful new tactics (especially in the group theory library).

On the debit side, there are a number of points to make. Most seriously, the treatment of tactics does not compare well with Isabelle. In HOL, a tactic either fails, or gives a single outcome, in contrast with

Isabelle, where multiple outcomes are common. Backtracking is not possible within HOL. There is also no mechanism for carrying out answer extraction (which is handled in Isabelle by scheme variables). This is a pity; answer extraction is very useful for reasoning about programs.

Much proof hacking is involved with manipulating a large number of assumptions, and carrying out such operations as rewriting them against each other or with given theorems. The built-in tactics such as POP_ASSUM and related tactics are not easy to work with, while the tactic RES_TAC seems to be quite limited in its power to solve goals if the assumptions are not in a very specific form. There is clearly a need for a range of fine_grained tactics for rewriting assumptions, and manipulating them in general. An experimental set of such tactics has been developed by Katherine Eastaughffe at DSTO. Although it is not difficult to write such tactics, it is time-consuming; such tactics should already be part of the system.

I think that HOL will only become really useful to a large number of people if the collection of tactics is completely overhauled. Presently there is a bewildering variety of tactics, with all sorts of odd names, some of which are high-powered useful tactics (such as REWRITE_TAC and its relatives), and others of which are highly specific "proof hacking" tactics written to solve a particular problem but not of general applicability. I think that it would be valuable to spell out some consistent naming conventions for tactics which people should try to stick to (a kind of allowed syntax for names), such as the prefixes ONCE, PURE etc. At the moment (to take a trivial example) we have ASM_REWRITE_TAC and POP_ASSUM, with different abbreviations used. Also, some older tactics will have been superseded by newer, more useful ones, and they should be deleted.[4]

8.1.5 Proof Management

The subgoal package allows great flexibility in the way theorems are proved. HOL can be used interactively (as a proof assistant) to develop proofs step by step. We can also use it as a proof checker by providing HOL with a complete proof and seeing whether HOL produces the required theorem as output.

There are problems with HOL's proof management. Proofs do not always look very 'natural'. Medium-sized to large-proofs can be quite laborious and time-consuming to construct, involving a good deal of proof 'exploration'. Often, a vast number of cases must be considered, and one then has the choice of defining a different tactic for each subgoal, or else using a single tactic as a blunt instrument on all the subgoals. Using a single tactic can improve the readability of the proof, but can be quite inefficient — some proofs can take several minutes, or even hours, of computation.

8.1.6 Instantiation of Types from Parent Theories

One of the most serious difficulties with HOL is that theories are inherited *en bloc*, and cannot be instantiated to specific instances. Essentially, what is needed is a mechanism for the refinement of types. For example: one might have a type (say 'colour') declared in a particular theory, and prove various things about it at a general level. Suppose then that we have a descendant of this theory, in which we want to instantiate this type to be (say) an enumerated type of the form {red, white, blue}. Presently this cannot be done in HOL, but it seems essential to have such a facility if HOL is to become really useful, especially in reasoning about mathematical theories which depend on some higher level abstract theory for many results. The theory of modular arithmetic, where group theory is used to prove facts about abelian subgroup of the integers, is a case in point. It has been studied by Elsa Gunter in the TUTORIAL [22]. She avoids the problem by defining predicates such as GROUP, SUBGROUP and NORMAL for arbitrary sets. Saying that a given set of integers is a subgroup, say, of the integers, amounts to saying that this predicate holds for this set. However, this method can be cumbersome in practice. Gunter has suggested an extension to HOL[28] in which theories can be instantiated, but the suggested enhancements are not widely accepted, and have not been implemented.

8.1.7 The Type Package

Tom Melham's type package is extremely useful: has taken a lot of the drudgery out of working with new compound data types in the style of ML recursive data types. However, the current package does have some limitations:

[4] Version 1.12 of HOL addresses some, but not all, of these criticisms.

1. It does not allow the definition of mutually recursive data types[5].
2. It does not allow the most general kind of recursive type (for example, one cannot define a recursive domain of values V such as

$$V = B + (V \rightarrow V)$$

in which functions from V to itself are "first-class objects", and can themselves be values. (Of course, we would not expect it to be easy to define such a domain!)
3. As for recursive functions, the only ones allowed are those which are recursive purely on one of their arguments. This is quite restrictive: it does not allow functions which are doubly recursive, or functions like

$$\mathrm{gcd}\; x\; y \;=\; (x = y \rightarrow x \;\square\; (x < y \rightarrow \mathrm{gcd}\; x\; y - x \;\square\; \mathrm{gcd}\; x - y\; y))$$

where the function is recursive on |x-y|.
4. Infix constructors are not allowed.

Clearly, formulating HOL theories which capture the formal semantics of programming languages will be much easier if the type package could be extended to cope with the above constructs.[6]

8.2 Comments on Isabelle

Our comments on Isabelle will necessarily be brief, because many have already been touched upon, and because most of the comments about the strengths of HOL apply equally well to Isabelle.

8.2.1 Ease of Use

Isabelle is much harder to learn than HOL, particularly because one must contend with a number of object logics. Writing new logics from scratch is quite difficult to grasp. However, generally Isabelle is a cleaner, more elegant and faster system than HOL (NB: new versions of HOL built on standard ML should not suffer from this comparison).

8.2.2 Object Logics and Theories

Isabelle is flexible in allowing a number of object logics, unlike the HOL system, in which the logic is fixed. There is, too, more flexibility in the ability to use more than one theory in a single session. Theories can be passed as arguments to various functions. Definitions can be overwritten in Isabelle theories; in HOL, however, one must quit and start again.

Isabelle's version of higher order logic is more cumbersome to use than the version present in the HOL system. Support for basic arithmetic is provided in Isabelle, but it is not as extensive as that in HOL.

Isabelle does not have anything equivalent to HOL's type definition package.

8.2.3 Libraries

Isabelle has as yet a modest collection of libraries, compared to those available in HOL. The main reason is that Isabelle does not yet have a large enough user base.

8.2.4 Tactics

Isabelle has full high-order unification (unlike HOL). Tactics can have more than one outcome. Backtracking and search are possible, and allow the construction of very powerful proof procedures. Scheme variables permit answer extraction in proofs — this is very useful.

Isabelle's meta-logic allows inference rules to be expressed and reasoning about them to be done in a uniform manner. This is made easy by a single tactic which carries out resolution in the meta-logic. However, in HOL, which has no meta-logic, one must write a new tactic for each new inference rule.

[5] A package written by a group at Aarhus does allow for mutually recursive types.

[6] The type package has been slightly improved in Version 1.12 of HOL.

8.3 Suggestions for Further Work

The work on implementing denotational semantics for imperative languages described in this paper can be extended in a number of ways. Maris Ozols and the author have constructed a HOL system for reasoning about a small imperative language [29]. Programs written using a simple, syntax are translated (along with specifications, if any) into HOL terms using a yacc-generated translator. Special purpose tactics allow interactive reasoning about programs, essentially using symbolic execution. A good deal of careful theorem proving is required for **while** loops in the system.

The work on operational semantics in Isabelle suggests a number of extensions, for example the study of

1. typed languages such as ML (with type inferencing)
2. imperative languages (study already begun)
3. process algebras such as Milner's Calculus of Communicating Systems (CCS) (an initial study begun)

A proof system for a significant subset of the Core language of Standard ML has been constructed by M Ozols and the author [30]. For future work, I believe that the study of process algebras offers the most challenge and will be the most rewarding; understanding concurrency remains a most difficult and long-term goal of computer science.

Chapter 9 Acknowledgments

+This paper was submitted as a thesis towards the degree of Graduate Diploma in Science at the Australian National University. I wish to thank my supervisor, Dr Malcolm Newey, Department of Computer Science, ANU, for suggesting the topic, introducing me to HOL, and for helpful discussions. I should also like to thank Professor Robin Stanton for his support and encouragement.

A large part of this work was carried out while I was a member of the Trusted Computer Systems Group of DSTO. I wish to express my gratitude to Dr Brian Billard, Head of Group, and to Mr Maris Ozols and Ms Katherine Eastaughffe for their interest in this work, and for useful discussions.

Bibliography

[1] L. E. Moser and P. M. Melliar-Smith. Formal Verification of Safety-Critical Systems. *Software-Practice and Experience*, 20:799, 1990.

[2] D. A. Schmidt. *Denotational Semantics — A Methodology for Language Development*. Allyn and Bacon, Boston, 1986.

[3] M. Nivat and J. C. Reynolds (eds). *Algebraic Methods in Semantics*. Cambridge University Press, 1985.

[4] M. J. C. Gordon. *The Denotational Description of Programming Languages*. Springer-Verlag, Berlin, 1979.

[5] J. E. Stoy. *Denotational Semantics: the Scott-Strachey approach to programming language theory*. MIT Press, 1977.

[6] P. D. Mosses. Denotational Semantics. *Handbook of Theoretical Computer Science*, page 577, 1990.

[7] C. Strachey. Towards a formal semantics. In *Working Conference on Formal Language Description Languages for Computer Programming*, pages 198–220. IFIP TC2, 1964.

[8] D. Bjorner and O. N. Oest (eds). *Towards a Formal Description of Ada*. Springer Verlag, Lecture Notes in Computer Science Vol 98, 1980.

[9] G. D. Plotkin. A Structural Approach to Operational Semantics. Report, University of Aarhus, Denmark.

[10] R. Milner. Language semantics. Notes for Computer Science 3 Course, University of Edinburgh, 1986.

[11] R. Roxas and M. C. Newey. Proof of Program Transformations. HOL '91 User Meeting, Aarhus, Denmark, Australian National University, 1991.

[12] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–583, 1969.

[13] P. Cousot. Methods and Logics for Proving Programs. *Handbook of Theoretical Computer Science (ed J van Leeuwen)*, page 843, 1990.

[14] R. Milner M. Gordon and C. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*. Lecture Notes in Computer Science, No 78. Springer-Verlag, 1979.

[15] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[16] A. Wikstrom. *Functional Programming Using Standard ML*. Prentice-Hall International Series in Computer Science, 1987.

[17] M. C. Newey and J. R. Ophel. ANU ML User's Manual. Technical report, Australian National University.

[18] L. C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge University Press, 1987.

[19] R. Cardell-Oliver. The Specification and Verification of Sliding Window Protocols. Computer Laboratory Technical Report 183, The University of Cambridge, 1989.

[20] J. J. Joyce. Using Higher Order Logic to Specify Computer Hardware and Architecture. In D. Edwards, editor, *Design Methodologies for VLSI and Computer Architecture*, pages 129–146. Procs. of the IFIP TC10 Working Conf. on Design Methodology in VLSI and Computer Architecture, Pisa, Italy, September 1988, North-Holland, 1989.

[21] A. Cant and K. Eastaughffe. The Application of Higher Order Logic to Security Models. Research Report ERL-0577-RR, Electronics Research Laboratory, DSTO, 1991.

[22] Cambridge Research Centre, SRI International and DSTO Australia. *The HOL System: DESCRIPTION, TUTORIAL and REFERENCE*, 1989.

[23] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[24] M. J. C. Gordon. Mechanizing Programming Logics in Higher Order Logic. In G. Birtwhistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 387–439. Springer-Verlag, 1989.

[25] L. C. Paulson and T. Nipkow. *Isabelle Tutorial and User's Manual*. Computer Laboratory, University of Cambridge, June 1990.

[26] L. C. Paulson. The Foundation of a Generic Theorem Prover. *Journal of Automated Reasoning*, 5:363–397, 1989.

[27] L. C. Paulson. Isabelle: The Next 700 Theorem Provers. *Logic and Computer Science (P Odifreddi, ed)*, pages 361–385, 1990.

[28] P. J. Windley. Abstracts from the HOL User Group Meeting. 1989.

[29] A. Cant and M. A. Ozols. The Role of Denotational Semantics in Program Verification. *Formal Aspects of Computing (to be submitted)*, 1992.

[30] A. Cant and M. A. Ozols. A Verification Environment for ML Programs, to be submitted to ACM Sigplan Workshop on ML, San Francisco, June, 1992.

APPENDIX A
Example: Denotational Semantics in HOL

```
% ------------------------------------------------------------ %
%           Implementation of Denotational                     %
%          Semantics for a simple Imperative Language          %
%            FILE:   impl.ml                                   %
%                                                              %
% see: Denotational Semantics, by D Schmidt p 76               %
% ------------------------------------------------------------ %
system 'rm impl.th';;
new_theory 'impl';;
map new_parent ['string'; 'digit';'decimal'];;
map loadf ['start_groups'; 'string_rules'];;
loadf '/packages/hol/sun3/tactics/local/odd';;
loadf '/packages/hol/sun3/tactics/local/asm';;


% ------------------------------------------------------------ %
%         Definitions of Semantic Algebras                     %
%               and  Basic Operations                          %
% ------------------------------------------------------------ %

% ------------------------------------------------------------ %
%             Identifiers                                      %
% ------------------------------------------------------------ %

new_type_abbrev ('Identifier', ":string");;

% ------------------------------------------------------------ %
%        Numbers (a lifted domain)                             %
%    Note: Truth Values should be lifted too but they are not  %
% ------------------------------------------------------------ %

let Number_Axiom = define_type 'Number'
  'Number =  number num |  undef_num';;

let Number_Induct = prove_induction_thm Number_Axiom;;

save_thm ('Number_one_one',  prove_constructors_one_one Number_Axiom);;
save_thm ('Number_distinct', prove_constructors_distinct Number_Axiom);;
save_thm ('Number_cases',    prove_cases_thm Number_Induct);;

let IS_NUMBER_DEF = new_recursive_definition false Number_Axiom
  'IS_NUMBER_DEF'
   "(is_number (number n) = T) /\
    (is_number undef_num = F) ";;

let IS_UNDEFINED_DEF= new_recursive_definition false Number_Axiom
  'IS_UNDEFINED_DEF'
   "(is_undefined (number n) = F) /\
    (is_undefined undef_num = T)  ";;

let GET_NUM_DEF = new_recursive_definition false Number_Axiom
    'GET_NUM_DEF'
   "(get_num (number n) = n) /\
    (get_num undef_num = 0) ";;

% ------------------------------------------------------------ %
%       The Store - includes an "undefined" or "error" element %
% ------------------------------------------------------------ %

let Store_Axiom = define_type 'Store'
  'Store = store (Identifier->Number) | undef_store';;

let Store_Induct = prove_induction_thm Store_Axiom;;

let Store_one_one =
  save_thm ('Store_one_one',  prove_constructors_one_one Store_Axiom);;

let Store_distinct =
  save_thm ('Store_distinct', prove_constructors_distinct Store_Axiom);;
```

```
let Store_cases =
  save_thm ('Store_cases',    prove_cases_thm Store_Induct);;

let Store_CASES_TAC t = DISJ_CASES_THEN
                        STRIP_ASSUME_TAC (SPEC t Store_cases) THEN
                        (ASM_REWRITE_TAC [Store_distinct]);;

let NEWSTORE_DEF = new_definition ('NEWSTORE_DEF',
  "(newstore:Store) =  store (\i. undef_num)");;

let ACCESS_DEF = new_recursive_definition false Store_Axiom
 'ACCESS_DEF'
  "(access i (store s)  = s i) /\
   (access i undef_store = undef_num)";;

let UPDATE_DEF = new_recursive_definition false Store_Axiom
 'UPDATE_DEF'
  "(update i v (store m)  = store (\j.(j=i) => v | m j)) /\
   (update i v undef_store = undef_store)";;



% ------------------------------------------------------------ %
%     if a store is proper, then updating it                   %
%     never leads to an undefined store                        %
% ------------------------------------------------------------ %

let LEMMA1 = prove_thm ('LEMMA1',
   "!s.~(s=undef_store) ==> ~(update i v s = undef_store)",
    GEN_TAC THEN STRIP_TAC THEN
    Store_CASES_TAC "s:Store" THEN
    REWRITE_TAC [UPDATE_DEF; Store_distinct] THEN
    TRIVIAL_TAC);;

let LEMMA2 = prove_thm ('LEMMA2',
  "~(update i v newstore = undef_store)",
  ASSUME_TAC (SPEC "store(\i:Identifier. undef_num)" LEMMA1) THEN
  ASSUME_TAC (SPEC "\i:Identifier. undef_num" Store_distinct) THEN
  RES_TAC THEN
  ASM_REWRITE_TAC [NEWSTORE_DEF]);;

% ------------------------------------------------------------ %
%         Syntax of the Language                               %
% ------------------------------------------------------------ %

let Expression_Axiom = define_type 'Expression'
  'Expression = Const Number |
                Var Identifier |
                Plus Expression Expression';;

let Expression_Induct = prove_induction_thm Expression_Axiom;;

save_thm ('Expression_one_one',
   prove_constructors_one_one Expression_Axiom);;

save_thm ('Expression_distinct',
   prove_constructors_distinct Expression_Axiom);;

save_thm ('Expression_cases',
  prove_cases_thm Expression_Induct);;

let BExpression_Axiom = define_type 'BExpression'
   'BExpression =
                True |
                False |
                Equals Expression Expression |
                Not BExpression';;

let BExpression_Induct = prove_induction_thm BExpression_Axiom;;

save_thm ('BExpression_one_one',
   prove_constructors_one_one BExpression_Axiom);;

save_thm ('BExpression_distinct',
    prove_constructors_distinct BExpression_Axiom);;

save_thm ('BExpression_cases',
  prove_cases_thm BExpression_Induct);;

let Command_Axiom = define_type 'Command'
    'Command  =  Skip |
                 Val Identifier Expression |
                 If BExpression Command Command |
                 While BExpression Command |
                 Seq Command Command |
                 Diverge' ;;

let Command_Induct = prove_induction_thm Command_Axiom;;

save_thm ('Command_one_one',
   prove_constructors_one_one Command_Axiom);;

save_thm ('Command_distinct',
   prove_constructors_distinct Command_Axiom);;

save_thm ('Command_cases',
  prove_cases_thm Command_Induct);;

new_type_abbrev ('Program', ":Command");;

let SEQL_DEF = new_list_rec_definition ('SEQL_DEF',
      "(Seql [] = Skip) /\
       (Seql (CONS command sequence) = Seq command  (Seql sequence))");;
% ------------------------------------------------------------- %
%  Fixed Point Combinator     %
% ------------------------------------------------------------- %

new_constant ('FIX', ":(*->*)->*");;

let FIX_EQ = new_axiom ('FIX_EQ', "!f:*->*.FIX f = f (FIX f)");;

% ------------------------------------------------------------- %
%             Semantic Equations                                 %
% ------------------------------------------------------------- %

let EXPR_DEF = new_recursive_definition false Expression_Axiom
    'EXPR_DEF'
    "(EXPR (Const v) s = ((s = undef_store) => undef_num | v)) /\
     (EXPR (Var i) s = access i s) /\
     (EXPR (Plus e1 e2) s =
        (is_number (EXPR e1 s) /\ is_number (EXPR e2 s))
 => number (get_num (EXPR e1 s) + get_num (EXPR e2 s)) | undef_num)";;

let BOOL_EXPR_DEF = new_recursive_definition false BExpression_Axiom
   'BOOL_EXPR_DEF'
   "(BOOL_EXPR True s = T) /\
    (BOOL_EXPR False s = F) /\
    (BOOL_EXPR (Equals e1 e2) s = (EXPR e1 s = EXPR e2 s))  /\
    (BOOL_EXPR (Not b) s = ~(BOOL_EXPR  b s))";;

% NB: the function COMMAND is strict in its store argument  %

let COMMAND_DEF = new_recursive_definition false Command_Axiom
    'COMMAND_DEF'
    "(COMMAND Skip s = s) /\
     (COMMAND (Val i e) s =
        (s = undef_store) => undef_store | update i (EXPR e s) s) /\
     (COMMAND (If b c1 c2) s =
        (s = undef_store) => undef_store |
         ((BOOL_EXPR b s) => (COMMAND c1 s) | (COMMAND c2 s)))  /\
    (COMMAND (While b c) s=
         FIX (\f t.(BOOL_EXPR b t => f (COMMAND c t) | t)) s) /\
   (COMMAND (Seq c1 c2) s =
        (s = undef_store) => undef_store | COMMAND c2 (COMMAND c1 s)) /\
     (COMMAND Diverge s = undef_store)";;

let PROGRAM_DEF = new_definition ('PROGRAM_DEF',
    "PROGRAM (p:Program) = \n.let s = (update 'input' n newstore) in
```

```
      let s' = (COMMAND p s) in (access 'output' s')");;

close_theory ();;


% ------------------------------------------------------------ %
        impl_tac.ml                                            %
%       Basic Tactics for Reasoning in impl                    %
%                                                              %
% ------------------------------------------------------------ %

load_theory 'impl';;
map loadf ['start_groups'; 'string_rules'; 'digit';'decimal'];;
include_theory 'impl';;
loadf '/packages/hol/sun3/tactics/local/odd';;
loadf '/packages/hol/sun3/tactics/local/asm';;

% ------------------------------------------------------------ %
%                 Tactics Used in the Proofs                   %
% ------------------------------------------------------------ %

let STRING_TAC       =  CONV_TAC  (DEPTH_CONV string_EQ_CONV);;
let STRING_RULE      =  CONV_RULE (DEPTH_CONV string_EQ_CONV);;

let DEC_EQ_TAC       =  CONV_TAC (ONCE_DEPTH_CONV DEC_EQ_CONV);;
let DEC_EQ_RULE      =  CONV_RULE (ONCE_DEPTH_CONV DEC_EQ_CONV);;

let DEC_ADD_TAC      =  CONV_TAC (ONCE_DEPTH_CONV DEC_ADD_CONV);;
let DEC_ADD_RULE     =  CONV_RULE (ONCE_DEPTH_CONV DEC_ADD_CONV);;

% NB : DEPTH_CONV diverges here %

let FUN_EQ_TAC       =  CONV_TAC (DEPTH_CONV FUN_EQ_CONV);;
let FUN_EQ_RULE      =  CONV_RULE (DEPTH_CONV FUN_EQ_CONV);;

let Command_CASES_TAC t = DISJ_CASES_THEN
                            STRIP_ASSUME_TAC (SPEC t Command_cases) THEN
                            (ASM_REWRITE_TAC [Command_distinct]);;

let Store_CASES_TAC t = DISJ_CASES_THEN
                            STRIP_ASSUME_TAC (SPEC t Store_cases) THEN
                            (ASM_REWRITE_TAC [Store_distinct]);;


%  SIMPLIFY_TAC does most of the work in simplifying expressions %

let DEFS = [IS_UNDEFINED_DEF;
       IS_NUMBER_DEF;
       GET_NUM_DEF;
       ACCESS_DEF;
       UPDATE_DEF;
       NEWSTORE_DEF;
          Number_one_one;
          Number_distinct;
          Store_one_one;
          Store_distinct;
          ADD_CLAUSES; % takes care of trivial arithmetic %
          LEMMA2];;


let SIMPLIFY_TAC =
      REPEAT
         (CHANGED_TAC
          (ASM_REWRITE_TAC DEFS THEN
           BETA_TAC THEN
           STRING_TAC THEN
           num_EQ_TAC THEN
           ADD_TAC ORELSE
           ONCE_REWRITE_TAC [FIX_EQ]));;

let EXPR_TAC       =  REWRITE_TAC [EXPR_DEF] ;;
let BOOL_EXPR_TAC  =  REWRITE_TAC [BOOL_EXPR_DEF];;

let COMMAND_TAC    =  REWRITE_TAC [COMMAND_DEF; SEQL_DEF];;

let PROGRAM_TAC    =  REWRITE_TAC [LET_DEF; PROGRAM_DEF] THEN BETA_TAC;;

let RUN_TAC        =  PROGRAM_TAC THEN
                      COMMAND_TAC THEN
                      BOOL_EXPR_TAC THEN
                      EXPR_TAC;;
```

APPENDIX B
Example: Natural Semantics in Isabelle

```
(* ---------------------------------------------------------------- *)
(*                                                                  *)
(*         f3.ml:                                                   *)
(*         Operational Semantics for a Functional                   *)
(*         Language with Recursion                                  *)
(*                                                                  *)
(* ---------------------------------------------------------------- *)


Pretty.margin:=72;
goals_limit := 250;
print_depth 1;


val const_decs =
 [ (["Const"],  [Aterm] ---> Aterm), (* expressions *)
   (["Var"],  [Aterm] ---> Aterm),
   (["Pr"],  [Aterm, Aterm] ---> Aterm),
   (["Apply"],  [Aterm, Aterm] ---> Aterm),
   (["If"],  [Aterm, Aterm, Aterm] ---> Aterm),
   (["Fn"],  [Aterm, Aterm] ---> Aterm),
   (["Let"],  [Aterm, Aterm] ---> Aterm),

   (["Val"],  [Aterm, Aterm] ---> Aterm),    (* declarations *)
   (["Comp"],  [Aterm, Aterm] ---> Aterm),
   (["And"],  [Aterm, Aterm] ---> Aterm),
   (["Rec"],  [Aterm] ---> Aterm),

   (["constant"], Atype),
   (["env"],  Atype),
   (["ide"],  Atype),
   (["value"],  Atype),
   (["expr"],  Atype),
   (["decl"],  Atype),

   (["empty"],  Aterm),            (* environment constructors *)
   (["bind"],  [Aterm, Aterm, Aterm] ---> Aterm),
   (["lookup"], [Aterm, Aterm, Aterm] ---> Aform), (* auxiliary fns *)
   (["combine"], [Aterm, Aterm, Aterm] ---> Aform),
   (["unfold"], [Aterm, Aterm, Aterm] ---> Aform),

   (["val"],  [Aterm] ---> Aterm), (* value constructors *)
   (["pair"],  [Aterm, Aterm] ---> Aterm),
   (["closure"], [Aterm, Aterm, Aterm, Aterm] ---> Aterm),

   (["const"],  [Aterm] ---> Aterm),
   (["tt"],  Aterm),   (* constants *)
   (["ff"],  Aterm),
   (["plus"],  Aterm),
   (["minus"],  Aterm),
   (["times"],  Aterm),
   (["zero"],  Aterm),

   (["apply"],  [Aterm, Aterm, Aterm] ---> Aform),

   (["seq"],  [Aterm, Aterm, Aterm] ---> Aform),   (* expr sequent *)
   (["seq'"],  [Aterm, Aterm, Aterm] ---> Aform),   (* decl sequent *)
   (["equiv"],  [Aterm,Aterm] ---> Aform) ];  (* expression equivalence  *)

val f3_thy  =
  extend_theory arith_thy "f3"
  (["env", "ide", "value", "expr", "decl", "constant"], const_decs)
  [ ("Const_type",
       "c : constant ==> Const (c) : expr"),
    ("Var_type",
       "x : ide ==> Var (x) : expr"),
    ("Pr_type",
       "[| e1 : expr; e2 : expr |] ==> Pr (e1,e2) : expr"),
    ("Apply_type",
   "[| e1 : expr; e2 : expr |] ==> Apply (e1, e2) : expr"),
("If_type",
   "[| e1 : expr; e2 : expr; e3 : expr |] ==> If(e1,e2,e3) : expr"),
("Fn_type",
   "[| x : ide; e : expr |] ==> Fn(x,e) : expr"),
("Let_type",
   "[| d : decl; e : expr |] ==> Let (d,e) : expr"),
("Val_type",
   "[| x : ide; e : expr |] ==> Val (x,e) : decl"),
("Comp_type",
   "[| d1 : decl; d2 : decl |] ==> Comp (d1, d2) : decl"),
("And_type",
   "[| d1 : decl; d2 : decl |] ==> And (d1, d2) : decl"),
("Rec_type",
   "d : decl ==> Rec (d) : decl"),
("tt_type",
   "tt : constant"),
("ff_type",
   "ff : constant"),
("plus_type",
   "plus : constant"),
("minus_type",
   "minus : constant"),
("times_type",
   "times : constant"),
("zero_type",
   "zero : constant"),
("empty_type",
   "empty : env"),
("bind_type",
   "[| E : env; x : ide; v : value |] ==> bind (x, v, E) : env"),
("val_type",
   "c : constant ==> val (c) : value"),
("const_type",
   "n : nat ==> const (n) : constant"),
("pair_type",
   "[| v1 : value; v2 : value |] ==> pair (v1,v2) : value"),
("closure_type",
   "[| x : ide; e : expr; E : env; E' : env |] ==> \
   \   closure (x,e,E,E') : value"),
("Const_rule",
   "[| E : env; c : constant |] ==> seq (E, Const (c), val (c))" ),
("Var_rule",
   "[| E : env; x : ide; v : value ; lookup (x, E, v) |] \
   \ ==> seq (E, Var (x), v)"),
("Pr_rule",
   "[| E : env; e1 : expr; e2 : expr; v1 : value; v2 : value; \
   \ seq (E,e1,v1); seq (E,e2,v2) |] ==> \
   \ seq (E, Pr(e1,e2), pair(v1,v2))"),
("Apply1_rule",
   "[| E0 : env; E : env; E' : env; E'' : env;  E''' : env; \
   \    e : expr; e1 : expr; e2 : expr; \
   \    v : value; v' : value; \
   \   seq (E0, e1, closure(x,e,E,E')); seq (E0,e2,v);     \
   \   unfold (E',E',E''); combine (E, E'', E''') ;   \
   \   seq (bind(x,v,E'''), e, v') |]    \
   \ ==> seq (E0, Apply(e1,e2),v')"),
("Apply2_rule",
   "[| E : env; e1 : expr; e2 : expr; \
   \    v : value; v' : value;  c : constant; \
   \   seq (E, e1, val(c)); seq (E,e2,v);     \
   \   apply (val(c),v,v') |]    \
   \ ==> seq (E, Apply(e1,e2),v')"),
("If1_rule",
   "[| E : env; e1 : expr; e2 : expr; e3 : expr;  \
   \ seq (E,e1,val(tt)); seq(E,e2,v) |] \
   \ ==> seq(E,If(e1,e2,e3),v)"),
("If2_rule",
   "[| E : env; e1 : expr; e2 : expr; e3 : expr;  \
   \ seq (E,e1,val(ff)); seq(E,e3,v) |] \
   \ ==> seq(E,If(e1,e2,e3),v)"),
("Fn_rule",
   "[| E : env; x : ide; e : expr |] ==> \
   \  seq(E, Fn(x,e), closure(x,e,E,empty))"),
 ("Let_rule",
   "[| E : env; E' : env; E'' : env; d : decl; e : expr; \
      seq' (E, d, E');  combine (E, E', E'');  \
   \  seq (E'', e, v) |] ==> seq (E, Let (d,e), v)"),
 ("Val_rule",
   "[| E : env; x : ide; e : expr; seq (E, e, v) |] \
      ==> seq' (E, Val(x,e), bind (x,v,empty))"),
 ("Comp_rule",
   "[| E : env; E' : env; E1 : env; E2 : env; E3 : env; \
   \  d1 : decl; d2 : decl ; \
   \  seq' (E , d1, E1); combine (E,E1,E'); \
   \  seq' (E', d2, E2); combine (E1,E2,E3) |] ==> \
   \  seq' (E, Comp (d1,d2), E3)"),
 ("And_rule",
   "[| E : env; E' : env; E1 : env; E2 : env; E3 : env; \
   \  d1 : decl; d2 : decl ; \
   \  seq' (E , d1, E1); seq' (E, d2, E2); combine (E1,E2,E3) |] ==>
   \  seq' (E, And (d1,d2), E3)"),
 ("Rec_rule",
   "[| E : env; E' : env; E'' : env; d : decl ; \
   \   unfold (E,E',E''); seq'(E,d,E') |] ==> seq' (E, Rec(d),E'')"),
 ("lookup1_rule",
   "[| E : env; x : ide;  v : value |] ==> \
    lookup (x, bind (x, v, E), v)" ),
 ("lookup2_rule",
   "[| E : env; x : ide; y : ide; v : value ;  \
   \ w : value;  ~(y = x : ide); lookup (x, E, v) |] \
   \ ==> lookup (x, bind (y, w, E), v)" ),
 ("combine_empty",
   "E : env ==> combine (E, empty, E)"),
 ("combine_bind",
   "[| E1 : env; E2 : env; E3 : env; x : ide; a : value;\
   \  combine (E1,E2,E3) |] ==>  \
      combine (E1, bind(x,a,E2), bind (x,a,E3))"),
 ("unfold_empty",
   "E : env ==> unfold (E, empty, empty)"),
 ("unfold_bind",
   "[| E : env; E1 : env; E2 : env; E' : env; E'' : env;
      f : ide; x : ide; e : expr; unfold (E,E',E'') |] ==>
      unfold (E, bind(f,closure(x,e,E1,E2),E'), \
      bind(f,closure(x,e,E1,E),E''))"),
 ("plus_rule",
   "[| m : nat; n : nat |] ==> \
    apply (val(plus), pair(val(const(m)),val(const(n))),  \
                val(const(m#+n)))"),
 ("minus_rule",
   "[| m : nat; n : nat |] ==> \
   \ apply (val(minus), pair(val(const(m)),val(const(n))),\
                val(const(m-n)))"),
 ("times_rule",
   "[| m : nat; n : nat |] ==> \ 
   \ apply (val(times), pair(val(const(m)),val(const(n))),\
                val(const(m#*n)))"),
 ("zero1_rule",
   "apply (val(zero), val(const(0)), val(tt))"),
 ("zero2_rule",
   "n : nat ==> apply (val(zero), val(Succ(n)), val(ff))"),
 ("equiv_def",
   "equiv (e1, e2) == ALL E : env. ALL v : value. \
   \ seq(E,e1,v) <-> seq(E,e2,v)")];


val Const_type   = get_axiom f3_thy "Const_type";
val Var_type     = get_axiom f3_thy "Var_type";
val Pr_type     = get_axiom f3_thy "Pr_type";
val Apply_type   = get_axiom f3_thy "Apply_type";
val If_type     = get_axiom f3_thy "If_type";
val Fn_type     = get_axiom f3_thy "Fn_type";
val Let_type     = get_axiom f3_thy "Let_type";
val Val_type     = get_axiom f3_thy "Val_type";
val Comp_type     = get_axiom f3_thy "Comp_type";
val And_type     = get_axiom f3_thy "And_type";
val Rec_type     = get_axiom f3_thy "Rec_type";
val empty_type   = get_axiom f3_thy "empty_type";
val tt_type  = get_axiom f3_thy "tt_type";
val ff_type  = get_axiom f3_thy "ff_type";
val plus_type  = get_axiom f3_thy "plus_type";
val minus_type  = get_axiom f3_thy "minus_type";
val times_type  = get_axiom f3_thy "times_type";
val zero_type  = get_axiom f3_thy "zero_type";
val bind_type  = get_axiom f3_thy "bind_type";
val val_type   = get_axiom f3_thy "val_type";
val const_type   = get_axiom f3_thy "const_type";
val pair_type   = get_axiom f3_thy "pair_type";
val closure_type  = get_axiom f3_thy "closure_type";

val Const_rule    = get_axiom f3_thy "Const_rule";
val Var_rule       = get_axiom f3_thy "Var_rule";
val Pr_rule       = get_axiom f3_thy "Pr_rule";
val Apply1_rule   = get_axiom f3_thy "Apply1_rule";
val Apply2_rule   = get_axiom f3_thy "Apply2_rule";
val If1_rule    = get_axiom f3_thy "If1_rule";
val If2_rule    = get_axiom f3_thy "If2_rule";
val Fn_rule    = get_axiom f3_thy "Fn_rule";
val Let_rule       = get_axiom f3_thy "Let_rule";
val Val_rule       = get_axiom f3_thy "Val_rule";
val Comp_rule       = get_axiom f3_thy "Comp_rule";
val And_rule       = get_axiom f3_thy "And_rule";
val Rec_rule       = get_axiom f3_thy "Rec_rule";

val lookup1_rule     = get_axiom f3_thy "lookup1_rule";
val lookup2_rule     = get_axiom f3_thy "lookup2_rule";
val combine_empty     = get_axiom f3_thy "combine_empty";
val combine_bind     = get_axiom f3_thy "combine_bind";
val unfold_empty     = get_axiom f3_thy "unfold_empty";
val unfold_bind     = get_axiom f3_thy "unfold_bind";
val plus_rule  = get_axiom f3_thy "plus_rule";
val minus_rule  = get_axiom f3_thy "minus_rule";
val times_rule  = get_axiom f3_thy "times_rule";
val zero1_rule  = get_axiom f3_thy "zero1_rule";
val zero2_rule  = get_axiom f3_thy "zero2_rule";
val equiv_def  = get_axiom f3_thy "equiv_def";

(* -------------------------------------------------------------------- *)
(* inference rules for sequents in the language                         *)
(* -------------------------------------------------------------------- *)

val expr_rules = [Const_rule, Var_rule, Pr_rule,  Apply1_rule, Apply2_rule,
                 If1_rule,  If2_rule, Fn_rule, Let_rule];

val decl_rules = [Val_rule, Comp_rule, And_rule, Rec_rule];

val lang_rules = decl_rules@expr_rules;

(* -------------------------------------------------------------------- *)
(* basic reduction rules                                                *)
(* -------------------------------------------------------------------- *)

val lookup_rules = [lookup1_rule, lookup2_rule];

val combine_rules = [combine_bind, combine_empty];

val unfold_rules = [unfold_bind, unfold_empty];

val basval_rules = [plus_rule, minus_rule,
                    times_rule, zero1_rule, zero2_rule];

(* -------------------------------------------------------------------- *)
(* simplification: basic logical, arithmetic and typing rules           *)
(* -------------------------------------------------------------------- *)

val HOL_rules = [refl, diff_0_eq_0, add_conv0,  diff_conv0,
        conj_intr,  disj_intr1, disj_intr2,
        iff_intr, True_intr];

val  HOL_type_rules = arith_type_rls@[Succ_type, Zero_type];

val f1_type_rules =
           [Const_type, Var_type, Pr_type, Apply_type, If_type, Fn_type,
             Let_type, Val_type, Comp_type, And_type, Rec_type,
      empty_type, bind_type, tt_type, ff_type,
      plus_type, minus_type, times_type, zero_type,
      val_type, const_type, pair_type, closure_type];

val common_asms = goal f3_thy
  "[| x : ide; y : ide; z : ide; f : ide; g : ide; \
   \ ~(x = y : ide); ~(y = x : ide);  \
   \ ~(x = z : ide); ~(z = x : ide);  \
   \ ~(x = f : ide); ~(f = x : ide);  \
   \ ~(x = g : ide); ~(g = x : ide);  \
   \ ~(y = z : ide); ~(z = y : ide);  \
   \ ~(y = f : ide); ~(f = y : ide);  \
   \ ~(y = g : ide); ~(g = y : ide);  \
   \ ~(f = z : ide); ~(z = f : ide);  \
   \ ~(g = z : ide); ~(z = g : ide);  \
   \ ~(f = g : ide); ~(g = f : ide);  \
   \ m : nat; n : nat;  \
   \ a : value; b : value; \
   \ e : expr; e' : expr;   E : env |] ==> T";


val simp_rules = common_asms @ HOL_rules @ HOL_type_rules @ f1_type_rules;
(* ----------------------------------------------------------------- *)
(*                                                                   *)
(*               tactics.ml                                          *)
(*          Useful tactics for proving programs in f3                *)
(*                                                                   *)
(* ----------------------------------------------------------------- *)

(* ----------------------------------------------------------------- *
 * Discriminators for subgoals.  These are used to make              *)
 * sure that only safe resolution rules are used                     *)
 * ----------------------------------------------------------------- *)

goal f3_thy "seq' (?E,?e,?v)";
val g = getgoal 1;
fun is_seq' t = could_unify (g, t);

goal f3_thy "seq' (?E,fixvar,?E)";
val g = getgoal 1;
fun is_safe_seq' t = (is_seq' t) andalso not (could_unify(g, t));

goal f3_thy "seq (?E,?e,?v)";
val g = getgoal 1;
fun is_seq t = could_unify (g, t);

goal f3_thy "seq (?E,fixvar,?E)";
val g = getgoal 1;
fun is_safe_seq t = (is_seq t) andalsq not (could_unify(g, t));

val is_sequent = fn t => (is_seq t) orelse (is_seq' t);

val is_safe_sequent = fn t => (is_safe_seq t) orelse (is_safe_seq' t);

goal f3_thy "combine (?E,?E,?E)";
val g = getgoal 1;
fun is_combine t = could_unify(g, t);

goal f3_thy "combine (?E,fixvar,?E)";
val g = getgoal 1;
fun is_safe_combine t = (is_combine t) andalso not (could_unify(g, t));

goal f3_thy "unfold (?E,?E,?E)";
val g = getgoal 1;
fun is_unfold t = could_unify(g, t);

goal f3_thy "unfold (?E,fixvar,?E)";
val g = getgoal 1;
fun is_safe_unfold t = (is_unfold t) andalso not (could_unify(g, t));

goal f3_thy "lookup (?x,?E,?v)";
val g = getgoal 1;
```

```
fun is_lookup t = could_unify(g, t);

goal f3_thy "lookup (?E,fixvar,?E)";
val g = getgoal 1;
fun is_safe_lookup t = (is_lookup t) andalso not (could_unify(g, t));

fun outgoal i  =
  fn state => let val (_,_,t,_) = dest_state(state,i) in t end;

(* -------------------------------------------------------------------- *)
(*    NOT_APPLICABLE  fails when there is a subgoal for which            *)
(*    a given selector is true, and succeeds otherwise                    *)
(* -------------------------------------------------------------------- *)

fun NOT_APPLICABLE selector =
  let fun tac (i,n, state) =
     if i>n then
        all_tac
     else
        if (selector (outgoal i state)) then no_tac else tac (i+1,n, state)
  in Tactic(fn state =>
       tapply(tac(1, length(prems_of state), state), state)) end;

(* -------------------------------------------------------------------- *)
(*    FIRSTONLY chooses the first subgoal for which the selector          *)
(*    function is true; it then applies the given tactic, and             *)
(*    fails otherwise                                                     *)
(* -------------------------------------------------------------------- *)

fun FIRSTONLY selector tf =
  let fun tac (i,n, state) =
     if i>n then
        no_tac
     else
        if (selector (outgoal i state)) then  tf(i) else tac (i+1,n, state)
  in Tactic(fn state =>
       tapply(tac(1, length(prems_of state), state), state)) end;

(* -------------------------------------------------------------------- *)
(*    reduction tactics                                                   *)
(*                                                                        *)
(*     lookup_tac: reduces using the lookup rules                         *)
(*                                                                        *)
(*    combine_tac: reduces using rules for                                *)
(*      combining environments                                            *)
(*                                                                        *)
(*    unfold_tac: reduces using rules for                                 *)
(*      unfolding environments                                            *)
(*                                                                        *)
(*    basval_tac: reduces using the rules for predefined                  *)
(*      constant functions                                                *)
(*                                                                        *)
(*    reduce_tac: performs a single reduction                             *)
(*                                                                        *)
(* -------------------------------------------------------------------- *)

val lookup_tac  =
   FIRSTONLY is_safe_lookup (resolve_tac lookup_rules );

val combine_tac =
   FIRSTONLY is_safe_combine (resolve_tac combine_rules);

val unfold_tac =
   FIRSTONLY is_safe_unfold (resolve_tac unfold_rules);

val basval_tac = FIRSTGOAL (resolve_tac basval_rules);

val reduce_tac  =
 lookup_tac ORELSE combine_tac ORELSE unfold_tac ORELSE basval_tac;

(* -------------------------------------------------------------------- *)
(*    step_tac: applies a language rule once, and safely                  *)
(* -------------------------------------------------------------------- *)
```

```
val step_tac  = FIRSTONLY is_safe_sequent (resolve_tac lang_rules);

(* ------------------------------------------------------------------ *)
(*    lang_tac: repeatedly applies the language rules,                *)
(*      simplifying after each step if possible                       *)
(*      It fails if any sequents remain                               *)
(* ------------------------------------------------------------------ *)

val lang_tac  = REPEAT1 (step_tac THEN TRY (REPEAT1 reduce_tac)) THEN
          NOT_APPLICABLE is_sequent;

(* ------------------------------------------------------------------ *)
(*    simp_tac: proves the goal by repeated assumptions               *)
(*      and use of the type information                               *)
(*      contained in the premises.                                    *)
(* ------------------------------------------------------------------ *)

val simp_tac = REPEAT (ares_tac simp_rules 1);

(* ------------------------------------------------------------------ *)
(*    f3_tac:  general-purpose tactic for proving goals by            *)
(*      depth-first search                                            *)
(* ------------------------------------------------------------------ *)

val f3_tac    = DEPTH_FIRST (has_fewer_prems 1)
                            (lang_tac THEN simp_tac);
```

APPENDIX C
Example: Translator for FUNC3

```
/* ----------------------------------------------------------- */
/*    translate.y                                               */
/*    yacc source code for converting between programs          */
/*    expressed in a simple functional language to              */
/*    suitable goals for Isabelle's HOL                         */
/* ----------------------------------------------------------- */
%{
#include "translate.h"
%}
%start text
%union {
  int num;
  String str;
  }
%token <str>  NUMBER IDENTIFIER PAIR APPLY IF THEN ELSE FN LET IN TRUE FALSE
%token <str>  VAL COMP AND REC COMMENT TURNSTILE EMPTY
%token <str>  ENV EVAL SCHEME ZERO ML
%type <str>  expr bexpr decl env bindlist
%left '+' '-'
%left '*'
%%

text:     /*  nothing */
        text comment
        text query
        text ml
     ;

comment: COMMENT   {  fprintf(outfile, "%s", $1); }
     ;

query: env TURNSTILE expr EVAL SCHEME
          { fprintf(outfile,
   "\ngoal f3_thy \n\"seq(%s, %s, %s) \"; by f3_tac;\n\n", $1, $3, $5); }
        env TURNSTILE decl EVAL SCHEME
          { fprintf(outfile,
   "\goal f3_thy \n\"seq'(%s, %s, %s) \"; by f3_tac; \n\n", $1, $3, $5); }
     ;

ml: ML    {  fprintf(outfile, "%s\n", $1); }
     ;

env:     '[' ']'    { $$ = make_nullop(EMPTY); }
     |  '[' bindlist ']'  { $$ = $2; }
     ;

bindlist:  '(' IDENTIFIER ',' NUMBER ')'
  { $$ = make_binop (ENV, $2, make_value(NUMBER,$4)); }
      |  bindlist ',' '(' IDENTIFIER ',' NUMBER ')'
          { $$ = make_ternop (ENV, $4, make_value(NUMBER,$6), $1); }
      ;


expr: NUMBER            { $$ = make_number ($1); }
     |  IDENTIFIER       { $$ = make_unop  (IDENTIFIER, $1);  }
     | expr ',' expr   { $$ = make_binop (PAIR, $1, $3); }
     | expr expr   { $$ = make_binop (APPLY, $1, $2); }
     |  IF bexpr THEN expr ELSE expr
         { $$ = make_ternop (IF, $2, $4, $6); }
     |  FN IDENTIFIER '.' expr  { $$ = make_binop(FN,$2,$4); }
     |  LET decl IN expr { $$ = make_binop(LET,$2,$4); }
     | expr '+' expr   { $$ = make_binop (APPLY, make_nullop('+'),
                                       make_binop (PAIR, $1, $3)); }
     | expr '*' expr   { $$ = make_binop (APPLY, make_nullop('*'),
                                       make_binop (PAIR, $1, $3)); }
     | expr '-' expr   { $$ = make_binop (APPLY, make_nullop('-'),
                                       make_binop (PAIR, $1, $3)); }
     |  '(' expr ')'   { $$ = $2; }
     ;
```

```
bexpr:  TRUE    { $$ = make_nullop (TRUE);  }
      | FALSE   { $$ = make_nullop (FALSE); }
      | ZERO expr   { $$ =make_binop (APPLY, make_nullop(ZERO),$2);}
      |   '(' bexpr ')'  { $$ = $2; }
      ;

decl:  VAL IDENTIFIER '=' expr  { $$ = make_binop (VAL, $2, $4); }
      | decl ';' decl  { $$ = make_binop (COMP, $1, $3); }
      | decl AND decl  { $$ = make_binop (AND, $1, $3); }
      | REC decl    { $$ = make_unop  (REC,$2); }
      | '(' decl ')'  { $$ = $2; }
      ;

%%
FILE *infile = stdin;
FILE *outfile = stdout;
int lineno;
main (argc, argv)
char * argv [];
 {
 yyparse();
 }

String make_number (n)
String n;
 {
 int bufsize = strlen(n)+20;
 String t = (String) malloc (bufsize);
 sprintf(t, "Const (const(%s))", n);
 return t;
 }

String make_value (c,s)
int c;
String s;
        {
        int bufsize = strlen(s)+20;
        String t = (String) malloc (bufsize);
        switch(c)
                {
                case NUMBER:
   sprintf(t, "Val (const(%s))", s);  break;
                }
        return t;
 }

String make_nullop (c)
int c;
        {
        int bufsize = 20;
        String t = (String) malloc (bufsize);
        switch(c)
                {
  case TRUE:
   sprintf(t ,"Const(tt)" ); break;
  case FALSE:
   sprintf(t ,"Const(ff)" ); break;
  case '+':
   sprintf(t ,"Const(plus)" ); break;
  case '*':
   sprintf(t ,"Const(times)" ); break;
  case '-':
   sprintf(t ,"Const(minus)" ); break;
  case ZERO:
   sprintf(t ,"Const(zero)" ); break;
  case EMPTY:
   sprintf(t ,"empty" ); break;
                }
         return t;
         }

String make_unop (c,s)
int c;
String s;
```

```
        {
        int bufsize = strlen(s)+20;
        String t = (String) malloc (bufsize);
        switch(c)
                {
                case IDENTIFIER:
   sprintf(t, "Var (%s)", s);  break;
                case REC:
   sprintf(t, "Rec (%s)", s);  break;
                }
        return t;
        }

String make_binop (c,t1,t2)
int c;
String t1, t2;
 {
 int bufsize = strlen(t1)+strlen(t2)+20;
 String t = (String) malloc (bufsize);
 switch(c)
  {
  case FN:
   sprintf(t ,"Fn (%s, %s)" , t1, t2); break;
  case APPLY:
   sprintf(t ,"Apply (%s, %s)" , t1, t2); break;
  case PAIR:
   sprintf(t ,"Pr (%s, %s)" , t1, t2); break;
  case LET:
   sprintf(t ,"Let (%s, %s)" , t1, t2); break;
  case VAL:
   sprintf(t ,"Val (%s, %s)" , t1, t2); break;
  case COMP:
   sprintf(t, "Comp (%s, %s)", t1, t2); break;
  case AND:
   sprintf(t, "And (%s, %s)", t1, t2); break;
  case ENV:
   sprintf(t, "bind (%s, %s, empty)", t1, t2); break;
  }
 return t;
 }

String make_ternop (c,t1,t2,t3)
int c;
String t1, t2, t3;
 {
 int bufsize = strlen(t1)+strlen(t2)+strlen(t3)+20;
 String t = (String) malloc (bufsize);
 switch(c)
  {
   case IF:
   sprintf(t, "If (%s, %s, %s)", t1, t2, t3); break;
   case ENV:
    sprintf(t, "bind(%s, %s, %s)", t1,t2,t3);
   }
 return t;
 }

yyerror(s)
char *s;
 {
 fprintf(stderr, " %s near line %d\n", s, lineno);
 }


/* ------------------------------------------------------------ */
/*    lex.l: lexical analyser                                   */
/* ------------------------------------------------------------ */

%{
extern int lineno;
#include "translate.h"
#include "y.tab.h"
%}
%%
```

```
[ \t]    { ; }
[\n]     { lineno++; }
"|-"     { return TURNSTILE ; }
"=>"     { return EVAL; }
"\\"     { return FN; }
"if"     { return IF; }
"then"     { return THEN; }
"else"      return ELSE; }
"let"    { return LET; }
"in"   { return IN; }
"and"    { return AND; }
"true"    { return TRUE; }
"false"    { return FALSE; }
"val"    { return VAL; }
"zero"    { return ZERO; }
"rec"    { return REC; }
"\\?"[a-zA-Z]  { yylval.str =
                            (String) malloc (yyleng);
     strcpy (yylval.str, yytext); return SCHEME; }
"(*.**)\n" { yylval.str =
                            (String) malloc (yyleng);
     strcpy (yylval.str, yytext);
     return COMMENT; }
"<<.*>>\n" { yylval.str =
                            (String) malloc (yyleng);
     strcpy (yylval.str, yytext + 2);
     yylval.str [yyleng-4] = '\0'; /* strip brackets */
     yylval.str [yyleng-5] = '\0';
     return ML; }
"Succ(0)"  { yylval.str = (String) malloc (yyleng);
     strcpy (yylval.str, yytext);
     return NUMBER; }
[mn0]    { yylval.str = (String) malloc (yyleng);
     strcpy (yylval.str, yytext);
     return NUMBER; }
[a-zA-Z][a-zA-Z0-9]* { yylval.str = (String) malloc (yyleng);
     strcpy (yylval.str, yytext);
     return IDENTIFIER; }
.    { return yytext [0]; }


 ***********************************************************/
 *                 translate.h                             */
 ***********************************************************/

#include <stdio.h>
typedef  char * String;

String make_number ();
String make_bool ();
String make_var ();
String make_value();
String make_binop ();
String make_ternop();
String mk_empty();
String make_env();
```

DISTRIBUTION

		Copy No.
Defence Science and Technology Organisation		
Chief Defence Scientist	)	
Central Office Executive	)	1
Counsellor, Defence Science, London		Cnt Sht
Counsellor, Defence Science, Washington		Cnt Sht
Scientific Adviser, Defence Central		1
Scientific Adviser, Defence Intelligence Organisation		1
Navy Scientific Adviser		1
Air Force Scientific Adviser		1
Scientific Adviser, Army		1
Electronics Research Laboratory		
Director		1
Chief, Communications Division		Cnt Sht
Chief, Electronic Warfare Division		Cnt Sht
Chief, Information Technology Division		1
Research Leader, Command and Control		1
Research Leader, Intelligence		1
PRSC3I		1
Head, Command Support Systems Group		1
Head, Information Systems Development Group		1
Head, Information Processing and Fusion Group		1
Head, Software Engineering Group		1
Head, Trusted Computer Systems Group		1
Head, Architectures Group		1
Head, VLSI Group		1
Head, Image Information Group		1
A. Cant (Author)		41
K. Eastaughffe		1
M. Ozols		1
S. Crawley		1
Publications and Component Support Officer		1
Graphics and Documentation Support		1
Libraries and Information Services		
Australian Government Publishing Service		1
Defence Central Library, Technical Reports Centre		1
Manager, Document Exchange Centre, (for retention)		1
National Technical Information Service, United States		2
Defence Research Information Centre, United Kingdom		2
Director Scientific Information Services, Canada		1
Ministry of Defence, New Zealand		1

National Library of Australia	1
Defence Science and Technology Organisation Salisbury, Main Library	2
Library Defence Signals Directorate, Melbourne	1
British Library Document Supply Centre	1
Spares	
Defence Science and Technology Organisation Salisbury, Main Library	6

Department of Defence

DOCUMENT CONTROL DATA SHEET

Page Classification
UNCLASSIFIED

Privacy Marking/Caveat
N/A

1a. AR Number AR-006-933	1b. Establishment Number ERL-0600-RR	2. Document Date January 1992	3. Task Number

4. Title

PROGRAM VERIFICATION USING HIGHER ORDER LOGIC

5. Security Classification

U	U	U
Document	Title	Abstract

S (Secret) C (Confi) R (Rest) U (Unclass)

- For UNCLASSIFIED docs with a secondary distribution LIMITATION, use (L) in document box.

6. No. of Pages 84

7. No. of Refs. –

8. Author(s)

A. Cant

9. Downgrading/Delimiting Instructions

N/A

10a. Corporate Author and Address

Electronics Research Laboratory
PO Box 1600
SALISBURY SA 5108

10b. Task Sponsor

DSD

11. Officer/Position responsible for

Security..........SOERL..........

Downgrading..........

Approval for Release..........DERL..........

12. Secondary Distribution of this Document

APPROVED FOR PUBLIC RELEASE

Any enquiries outside stated limitations should be referred through DSTIC, Defence Information Services, Department of Defence, Anzac Park West, Canberra, ACT 2600.

13a. Deliberate Announcement

No Limitation

13b. Casual Announcement (for citation in other documents)

☑ No Limitation

□ Ref. by Author , Doc No. and date only.

14. DEFTEST Descriptors

Computer program verification, High level languages, Semantics, Logic Programming

15. DISCAT Subject Codes

1205

16. Abstract

This paper describes a number of experiments in program verification carried out within two automated proof assistants, namely the HOL (Higher Order Logic) system and Isabelle. Various approaches to programming language semantics are described. Theories and tactics for proving the correctness of programs written in small functional and imperative languages are then constructed within HOL and Isabelle.

16. Abstract (CONT.)

17. Imprint

Electronics Research Laboratory
PO Box 1600
SALISBURY SA 5108

18. Document Series and Number	19. Cost Code	20. Type of Report and Period Covered
ERL-0600-RR	822522	RESEARCH REPORT

21. Computer Programs Used

Higher Order Logic
Isabelle

22. Establishment File Reference(s)

23. Additional information (if required)