Reasoning About Software

Roger Hale
SRI International
Cambridge Computer Science Research Centre
(email: rwsh@cam.sri.com)

Abstract

Although HOL *has mostly been used for hardware verification, it is an equally appropriate tool for reasoning about software. Many kinds of programming language semantics (denotational, algebraic, operational, etc.) may be embedded in the* HOL *logic. Based on these semantics, one can construct tools to support program verification, e.g. using a standard programming logic. Furthermore, by modelling the compilation process in the* HOL *logic, one can verify that a program's meaning is in fact preserved when it is compiled and run, thus bridging the gap between hardware and software.*

The general approach is illustrated for a simple sequential programming language, by describing its syntax and semantics, its Hoare Logic, and its compilation.

1 Introduction

HOL is a recognised leader in the field of hardware verification, but has not been nearly so widely used for software. This situation is now changing, however, as more effort is put into tools for program verification, and it is found that HOL has a great deal to offer in this area.

1.1 Why Use HOL?

There are a number of reasons why one might choose to use HOL for reasoning about programs; they mostly have to do with its generality.

- *Expressive power.* Just about any concept you might need for reasoning about software can be defined in the HOL logic. Many programming concepts are naturally expressed as higher-order functions.
- *Mathematical basis.* The HOL system supports natural mathematical reasoning, with powerful tools for inductive definition and proof.
- *Ease of extension.* HOL is programmable; if the tools you need are not there, you can construct them. Thus, special purpose logics can be implemented.
- *Security.* Provided that definitional principles are followed, all theorems are deduced from a handful of axioms by a small number of primitive rules.

For these reasons HOL is an environment well suited to embedding different software verification techniques. Many different methods and theories can be brought to bear on a single problem. For example, in the course of reasoning about real programs one might use, in addition to a program logic or process algebra, theories of arithmetic or lists in order to reason about data structures.

Furthermore, by modelling the compilation process in the HOL logic, one can verify that a program's meaning is preserved when it is compiled and run. In this way, HOL can be used to bridge the gap between hardware and software.

1.2 The General Approach

An embedding of a programming language and its theory in HOL may be performed in three steps:

- *Syntax.* The first step is to define a representation for programs in HOL, which is most conveniently achieved using the built-in package for defining recursive types [Mel88]. It is then possible to treat programs as 'objects' and their structure may be exploited in proofs, using case analysis, structural induction, and so on.

0-8186-2460-4/92 $03.00 © 1992 IEEE

- *Semantics.* The purpose here is to construct a model of program behaviour. There are many ways to do this; for example, the model may be based on denotational or operational semantics, traces or temporal logic. The choice is governed largely by what aspects of behaviour are considered important.
- *Reasoning support.* In order to use the model developed in the first two steps to reason about programs, one needs to prove some basic theorems and define special purpose tools for manipulating programs and specifications. These may implement any combination of programming logics, algebraic laws, temporal logic, refinement, etc.

Most of the work is in the third step. Indeed, it is possible to omit the semantics (and even the syntax) definition altogether by axiomatising the theory. This approach has advantages: it eliminates the dependence on a particular model and it reduces the burden of proof; but there is no check on the introduction of new theorems and experience shows that it is all too easy to introduce inconsistencies.

To illustrate the general approach, and to give some substance to the claim that HOL is a suitable environment for reasoning about software, this tutorial focusses on a simple sequential programming language, describing its syntax and semantics, its Hoare Logic, and its compilation. Much of this presentation is based on Gordon's embedding of programming logics in HOL [Gor89], which is supplied as a system library.

2 Syntax

The language chosen for illustration contains skip and assignment commands, and constructors for sequence, conditional and while loop. Expressions are either numeric or boolean; all variables have numeric values, booleans are only used in conditions. Defining the syntax using the recursive type definition package is quite intuitive and fairly painless.

2.1 Expressions

The type **nexp** of numeric expressions contains constants whose values are taken from the the type num of natural numbers, variables whose names are taken from the type **string** of ASCII strings, and binary operators from the type num → num → num of binary operators on numbers. It is defined as follows

```
nexp ::= CON num
       | VAR string
       | NOP (num → num → num) nexp nexp
```

Similarly, the type **bexp** of boolean expressions contains the logical constants, the negation operator, binary boolean operators lifted from the type bool → bool → bool, and binary relations on numeric expressions from the type num → num → bool of relations over the natural numbers.

```
bexp ::= TRUE
       | FALSE
       | NOT bexp
       | BOP (bool → bool → bool) bexp bexp
       | ROP (num → num → bool) nexp nexp
```

In practice, not every binary operation will be an allowed expression, nor will every string be an allowed variable name, but these restrictions may be dealt with by defining a parser and pretty-printer, using the HOL system tools [vT90, Bou90], to convert between the actual language syntax and its HOL representation. Such tools can also be used to overcome the awkwardness of the HOL syntax.

2.2 Commands

The type comm of commands, contains skip, assignment, sequential composition, conditional and while loop; It is defined as follows:

```
comm ::= SKIP
       | ASSIGN string nexp
       | SEQ comm comm
       | IF bexp comm comm
       | WHILE bexp comm
```

Each type definition generates a theorem defining the type operators, from which a number of other useful theorems may be proved automatically. For example, the structural induction theorem for commands, given below, is generated from the type definition comm by means of the meta-level tool, `prove_induction_theorem`.

```
∀P•
  P SKIP ∧
  (∀x e • P(ASSIGN x e)) ∧
  (∀c1 c2 • P c1 ∧ P c2 ⟹ P(SEQ c1 c2)) ∧
  (∀c1 c2 • P c1 ∧ P c2 ⟹ ∀b • P(IF b c1 c2)) ∧
  (∀c • P c ⟹ ∀b • P(WHILE b c)) ⟹
  (∀c • Pc)
```

This theorem states that a predicate P is true for all commands ($\forall c \bullet P\ c$) if it is true recursively over the structure. For example, if P holds for two commands, (P c1) and (P c2), then it also holds for their sequential composition, P(SEQ c1 c2). This theorem forms the basis of a tactic for proof by structural induction.

3 Semantics

There are many ways to represent programming language semantics. This section concentrates on a particular denotational semantics for the language defined above, but any number of other approaches would serve equally well. For example, Melham's inductive definition package [Mel91] makes easy work of defining an operational semantics; and a semantics based on Interval Temporal Logic [Mos86, Hal89] can be used to model real-time and parallel behaviours.

3.1 Expressions

The model is to be state-based, where a state is a mapping from variable names to values of type $\mathtt{string} \rightarrow \mathtt{num}$. In HOL,

$$\mathtt{state} \triangleq \mathtt{string} \rightarrow \mathtt{num}$$

Each expression is assumed to be evaluated on a single state, and is therefore represented as a function from states to values. The meaning (N e s) of numeric expression e on state s is defined recursively over the type nexp in the obvious way,

$$\begin{array}{lcl} (\mathtt{N}\ (\mathtt{CON}\ \mathtt{n})\ \mathtt{s} & \triangleq & \mathtt{n}) \wedge \\ (\mathtt{N}\ (\mathtt{VAR}\ \mathtt{x})\ \mathtt{s} & \triangleq & \mathtt{s\ x}) \wedge \\ (\mathtt{N}\ (\mathtt{NOP}\ \mathtt{nop}\ \mathtt{e1}\ \mathtt{e2})\ \mathtt{s} & \triangleq & \mathtt{nop}\ (\mathtt{N\ e1\ s})(\mathtt{N\ e2\ s})) \end{array}$$

and similarly, the meaning (B b s) of boolean expression b on state s is defined recursively over the type bexp.

$$\begin{array}{lcl} (\mathtt{B\ TRUE\ s} & \triangleq & \mathtt{T}) \wedge \\ (\mathtt{B\ FALSE\ s} & \triangleq & \mathtt{F}) \wedge \\ (\mathtt{B}\ (\mathtt{NOT\ b})\ \mathtt{s} & \triangleq & \neg(\mathtt{B\ b\ s})) \wedge \\ (\mathtt{B}\ (\mathtt{BOP\ bop\ b1\ b2})\ \mathtt{s} & \triangleq & \mathtt{bop}\ (\mathtt{B\ b1\ s})(\mathtt{B\ b2\ s})) \wedge \\ (\mathtt{B}\ (\mathtt{ROP\ rop\ e1\ e2})\ \mathtt{s} & \triangleq & \mathtt{rop}\ (\mathtt{N\ e1\ s})(\mathtt{N\ e2\ s})) \end{array}$$

3.2 Commands

In the semantics described below each command is modelled as a relation over pairs of states, $(\mathtt{s}, \mathtt{s}')$, where s is the initial state and s′ the final state. The type of this representation will be called csem,

$$\mathtt{csem} \triangleq (\mathtt{state} \times \mathtt{state}) \rightarrow \mathtt{bool}$$

Note that this model is only useful for terminating commands, since there must be a final state; a non-terminating loop is simply false.

The skip command has no effect on the state; its meaning is given by the relation Skip, of type csem, defined as follows.

$$\mathtt{Skip}(\mathtt{s}, \mathtt{s}') \triangleq (\mathtt{s}' = \mathtt{s})$$

The assignment of expression e to variable x holds if the final state is the same as the initial state, but with the value of e substituted for x; that is, $\mathtt{s}' = \mathtt{s}[(\mathtt{e\ s})/\mathtt{x}]$. The substitution s[e/x] is represented as BND e x s,

$$\mathtt{BND\ e\ x\ s\ z} \triangleq (\mathtt{z} = \mathtt{x}) \Rightarrow \mathtt{e\ s} \mid \mathtt{s\ z}$$

and the assignment is captured by the relation

$$(\mathtt{x} := \mathtt{e})(\mathtt{s}, \mathtt{s}') \triangleq (\mathtt{s}' = \mathtt{BND\ e\ x\ s})$$

The sequential composition of two commands c1 and c2 holds if there is an intermediate state t such that it is possible to get from s to t by c1 and from t to s′ by c2.

$$(\mathtt{c1}; \mathtt{c2})(\mathtt{s}, \mathtt{s}') \triangleq \exists \mathtt{t} \bullet \mathtt{c1}(\mathtt{s}, \mathtt{t}) \wedge \mathtt{c2}(\mathtt{t}, \mathtt{s}')$$

The conditional command simply behaves like the first or second branch according to the value of the condition.

$$(\mathtt{If\ b\ c1\ c2})(\mathtt{s}, \mathtt{s}') \triangleq \mathtt{b\ s} \Rightarrow \mathtt{c1}(\mathtt{s}, \mathtt{s}') \mid \mathtt{c2}(\mathtt{s}, \mathtt{s}')$$

Finally, the while loop may be defined in the usual way as the least fixed point of F, where $\mathtt{F\ X} = \mathtt{If\ b}\ (\mathtt{c}; \mathtt{X})\ \mathtt{Skip}$.

$$\mathtt{While\ b\ c} \triangleq \mathtt{FIX}(\lambda\, \mathtt{X} \bullet \mathtt{If\ b}\ (\mathtt{c}; \mathtt{X})\ \mathtt{Skip})$$

The definition of the least fixed point operator, FIX, can be found, together with supporting theory, in the HOL library theory of fixed points.

With these definitions, the meaning (C c) of command c may be defined recursively over the type comm.

$$
\begin{array}{lcl}
\texttt{(C SKIP} & \triangleq & \texttt{Skip)} \wedge \\
\texttt{(C (ASSIGN x e)} & \triangleq & \texttt{x := (N e))} \wedge \\
\texttt{(C (SEQ c1 c2)} & \triangleq & \texttt{(C c1);(C c2))} \wedge \\
\texttt{(C (IF b c1 c2)} & \triangleq & \texttt{If (B b) (C c1) (C c2))} \wedge \\
\texttt{(C (WHILE b c)} & \triangleq & \texttt{While (B b) (C c))}
\end{array}
$$

This completes the embedding of the language; all that remains is to build some reasoning tools around it.

4 Reasoning

The subject of reasoning about programs is large and I cannot possibly do justice to it in this tutorial. Instead, I will concentrate on a simple method, the Hoare Logic of partial correctness, for the simple language defined above. My description is based on Gordon's implementation of programming logics in HOL [Gor89].

4.1 Hoare Logic

The first thing is to relate the pre/post-condition style of specification, {p}c{q}, relates to the programming language semantics. This is straightforward for the semantics described above — whenever the assertion p is true of state s and the command c changes the state from s to s′, then the assertion q must hold of s′.

The specification {p}c{q} is represented in HOL by the predicate SPEC.

$$\texttt{SPEC(p,c,q)} \triangleq \forall \texttt{s s'} \bullet \texttt{p s} \wedge \texttt{c(s,s')} \Longrightarrow \texttt{q s'}$$

I shall also need some other new operators for dealing with assertions. The 'command' Assert p is used to embed assertions within programs,

$$\texttt{(Assert p)(s,s')} \triangleq \texttt{(p s)} \wedge (\texttt{s'} = \texttt{s})$$

the logical connectives ~, /\ and => are used to combine assertions

$$
\begin{array}{lcl}
\texttt{(\~{}p)s} & \triangleq & \neg\texttt{(p s)} \\
\texttt{(p /\textbackslash\ q)s} & \triangleq & \texttt{(p s)} \wedge \texttt{(q s)} \\
\texttt{(p => q)s} & \triangleq & \texttt{(p s)} \Longrightarrow \texttt{(q s)}
\end{array}
$$

and the operator Valid is used to state the validity of assertions,

$$\texttt{Valid p} \triangleq \forall \texttt{s} \bullet \texttt{p s}$$

In the system implementation of Hoare Logic, there is a certain amount of trickery going on behind the scenes to convert between boolean terms (of type bool) and assertions (of type state → bool), so that assertions appear as ordinary HOL booleans and these higher-order operators are not needed. For simplicity of presentation, I have chosen not to do that here.

4.2 Rules

The next step is to verify that the rules of Hoare Logic do in fact hold for the given semantics. This is done by proving a theorem corresponding to each rule — a rule in HOL is a meta-language program and cannot be verified directly. Each theorem takes the form of an implication, whose antecedent is a conjunction of terms corresponding to the top of the rule, together with any side conditions, and whose conclusion corresponds the term under the rule.

The rules and corresponding theorems for Hoare Logic are listed in figure 1. The rules themselves are implemented as ML functions based on these theorems. For example, the rule for the conditional uses *modus ponens* on the conditional theorem, matching the antecedent of the implication with the actual code to generate the instantiated form of the conclusion. Most of the rules are like this, though the assignment rule also requires some simplification of the substituted expression.

4.3 Tactics

These rules can then be used to construct tactics for goal-directed proof. The tactics are again written as ML functions, and work in the usual way by decomposing a goal of the form {p}c{q} into one or more simpler subgoals whose verification will establish {p}c{q}.

The assignment tactic decomposes an assignment specification into an implication which entails the specification, if it can be proved.[1]

$$\frac{\texttt{\{p\}x := e\{q\}}}{\texttt{Valid(p => q o (BND e x))}}$$

The tactic for decomposing sequential composition works by either stripping trailing assignments, or

[1] I shall use the notation

$$\frac{\texttt{g}}{\texttt{sg}_1, \texttt{sg}_2, \ldots, \texttt{sg}_n}$$

to mean: 'to prove the goal g, it is sufficient to prove the subgoals $\texttt{sg}_1, \texttt{sg}_2, \ldots \texttt{sg}_n$'.

Rule		HOL theorem
Skip	$\frac{}{\{p\}\texttt{Skip}\{p\}}$	$\forall p \bullet \texttt{SPEC}(p, \texttt{Skip}, p)$
Assignment	$\frac{}{\{p[e/x]\}(x := e)\{p\}}$	$\forall p\ x\ e \bullet \texttt{SPEC}(p \circ (\texttt{BND}\ e\ x), x := e, p)$
Sequence	$\frac{\{p\}c1\{q\} \quad \{q\}c2\{r\}}{\{p\}(c1; c2)\{q\}}$	$\forall p\ q\ r\ c1\ c2 \bullet$ $\texttt{SPEC}(p, c1, q) \wedge \texttt{SPEC}(q, c2, r) \Longrightarrow$ $\texttt{SPEC}(p, c1; c2, r)$
Conditional	$\frac{\{p /\backslash b\}c1\{q\} \quad \{p /\backslash \sim b\}c2\{q\}}{\{p\}(\texttt{If}\ b\ c1\ c2)\{q\}}$	$\forall p\ q\ c1\ c2\ b \bullet$ $\texttt{SPEC}(p /\backslash b, c1, q) \wedge \texttt{SPEC}(p /\backslash \sim b), c2, q) \Longrightarrow$ $\texttt{SPEC}(p, \texttt{If}\ b\ c1\ c2, q)$
While	$\frac{\{p /\backslash b\}c\{p\}}{\{p\}(\texttt{While}\ b\ c)\{p /\backslash \sim b\}}$	$\forall p\ c\ b \bullet$ $\texttt{SPEC}(p /\backslash b, c, p) \Longrightarrow$ $\texttt{SPEC}(p, \texttt{While}\ b\ c, p /\backslash \sim b)$
Pre-cond. strengthening	$\frac{\{p\}c\{q\} \quad \texttt{Valid}(p' \Rightarrow p)}{\{p'\}c\{q\}}$	$\forall p\ p'\ q\ c \bullet$ $\texttt{Valid}(p' \Rightarrow p) \wedge \texttt{SPEC}(p, c, q) \Longrightarrow$ $\texttt{SPEC}(p', c, q)$
Post-cond. weakening	$\frac{\{p\}c\{q\} \quad \texttt{Valid}(q \Rightarrow q')}{\{p\}c\{q'\}}$	$\forall p\ q\ q'\ c \bullet$ $\texttt{SPEC}(p, c, q) \wedge \texttt{Valid}(q \Rightarrow q') \Longrightarrow$ $\texttt{SPEC}(p, c, q')$

Figure 1: Hoare Logic in HOL.

looking for an assertion embedded in the program and breaking the program at that point.

$$\frac{\{p\}c1;\ldots;cn;x := e\{q\}}{\{p\}c1;\ldots;cn\{q[e/x]\}}$$

$$\frac{\{p\}c1;\ldots;cn;(\texttt{Assert}\ r);c\{q\}}{\{p\}c1;\ldots;cn\{r\} \quad \{r\}c\{q\}}$$

The conditional tactic breaks a conditional specification into two parts in the obvious way.

$$\frac{\{p\}\texttt{If}\ b\ c1\ c2\{q\}}{\{p /\backslash b\}c1\{q\} \quad \{p /\backslash \sim b\}c2\{q\}}$$

Finally, the tactic for decomposing the while loop looks for a loop invariant and, based on that, produces three subgoals.

$$\frac{\{p\}\texttt{While}\ b\ (\texttt{Assert}\ r;c)\{q\}}{\begin{array}{c}\texttt{Valid}(p \Rightarrow r)\\ \{r /\backslash b\}c\{r\}\\ \texttt{Valid}((r /\backslash \sim b) \Rightarrow q)\end{array}}$$

As mentioned above, the implementation of Hoare Logic that is provided as a system library contains a certain amount of HOL wizardry to make proofs appear more natural. In particular, pre- and post-conditions may be expressed as simple booleans, rather than relations over states, as I have them. Variables in pre- and post-conditions are then HOL variables and expressions are either boolean or numeric; conversion between these forms and the corresponding forms that appear in programs, where variables are strings and expressions functions of states, is carried out automatically.

5 Compilation

This final section looks at compilation, where software and hardware semantics meet. For illustration, I will use an abstract stack-based machine (adapted from [Gor91]) as it avoids much of the messiness associated with real compilation (such as the symbol table!). However, this does not imply

that such details cannot be handled in HOL. They can, as Joyce has shown [Joy89].

5.1 Machine Code

The first step towards specifying a compiler in HOL is to define the syntax and semantics of the assembly code. For our stack machine the following instruction syntax will be used:

```
inst ::=
  POP                          Pop top of stack
| OP0 num                      Push value onto stack
| OP1 num → num                Unary operation
| OP2 num → num → num          Binary operation
| GET string                   Push contents of memory
| PUT string                   Pop and store the result
| JMP num                      Unconditional jump
| JMZ num                      Pop stack, jump if zero
```

The semantics of instructions may be defined in the usual way using 'conventional' hardware semantics, as described by Gordon [Gor91]. Alternatively, taking the approach advocated by Hoare [Hoa90], the effect of each instruction may be represented in (the same way as) the high-level programming language. For example, the effect of the unconditional jump, `JMP n`, is to assign `n` to the program counter.

5.2 Compiler Specification

A compiler can be specified by a function or relation defined recursively over the syntax of commands and expressions. In the specification below the compilation of an expression or command is represented as a relation over an instruction store, which is a function of type **num** → **inst** from addresses to instructions, and two addresses of type **num**, which mark the beginning and end of the compiled code.

In this formulation, the relation `INST i ROM f t` is true if the instruction store `ROM` contains the single instruction `i` between addresses `f` and `t`.

$$(\texttt{INST i})\ \texttt{ROM f t} \triangleq (\texttt{ROM f} = \texttt{i}) \wedge (\texttt{t} = \texttt{f} + 1)$$

`ROM` contains the concatenation of two instruction sequences between `f` and `t` if there is an intermediate point `f + n` such that the first is stored between `f` and `f + n` and the second is stored between `f + n` and `t`.

$$(\texttt{c1 ++ c2})\ \texttt{ROM f t} \triangleq \exists \texttt{n} \bullet \texttt{c1 ROM f (f + n)} \wedge \texttt{c2 ROM (f + n) t}$$

The compilation of numeric expressions is then specified by the relation `CN`, as follows:

$$\begin{aligned}
&(\texttt{CN (CON n)} && \triangleq \texttt{INST(OP0 n))} \wedge \\
&(\texttt{CN (VAR x)} && \triangleq \texttt{INST(GET x))} \wedge \\
&(\texttt{CN (NOP nop e1 e2)} && \triangleq \\
&\quad \texttt{(CN e1) ++ (CN e2) ++ INST(OP2 nop))}
\end{aligned}$$

and likewise, the compilation of boolean expressions by the relation `CB`,

$$\begin{aligned}
&(\texttt{CB TRUE} && \triangleq \texttt{INST(OP0 1))} \wedge \\
&(\texttt{CB FALSE} && \triangleq \texttt{INST(OP0 0))} \wedge \\
&(\texttt{CB (NOT b)} && \triangleq \\
&\quad \texttt{(CB b) ++ INST(OP1 EQ0))} \wedge \\
&(\texttt{CB (BOP bop b1 b2)} && \triangleq \\
&\quad \texttt{(CB b1) ++ (CB b2) ++ INST(OP2(B2N bop)))} \wedge \\
&(\texttt{CB (ROP rop e1 e2)} && \triangleq \\
&\quad \texttt{(CN e1) ++ (CN e2) ++ INST(OP2(R2N rop)))}
\end{aligned}$$

where `EQ0`, `B2N` and `R2N` are the appropriate type conversions.

Compilation of the command `SKIP` generates no code; that is, the start and end points are the same.

$$\texttt{C\_SKIP ROM f t} \triangleq (\texttt{t} = \texttt{f})$$

Compilation of the assignment `ASSIGN x e` must generate code to evaluate the expression `e` followed by an instruction to store its value at `x`.

$$\texttt{C\_ASSIGN x e} \triangleq \texttt{e ++ INST(PUT x)}$$

Compilation of the conditional `IF b c1 c2` generates code to evaluate `b` followed by a conditional jump to the code for `c1` or `c2`, depending on the value of `b`.

$$\begin{aligned}
&(\texttt{C_IF b c1 c2})\ \texttt{ROM f t} \triangleq \\
&\ (\texttt{b ++} \\
&\quad (\lambda\ \texttt{ROM f t'} \bullet \\
&\qquad (\texttt{INST(JMZ t')} \texttt{ ++ c1 ++ INST(JMP t))} \\
&\qquad \texttt{ROM f t')} \texttt{ ++} \\
&\quad \texttt{c2)} \\
&\ \texttt{ROM f t}
\end{aligned}$$

The while loop `WHILE b c` also begins with code to evaluate `b`, followed by a conditional jump to the end if `b` is false. This is followed by code for `c` and then a jump back to the start.

$$\begin{aligned}
&\texttt{C_WHILE b c ROM f t} \triangleq \\
&\quad \texttt{(b ++ INST(JMZ t) ++ c ++ INST(JMP f))ROM f t}
\end{aligned}$$

The compilation of commands is therefore specified by the relation CC, as follows.

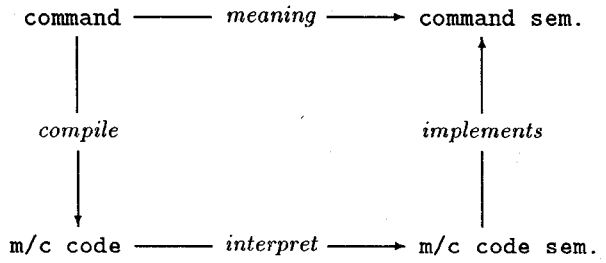
$$
\begin{array}{lcl}
(\texttt{CC SKIP} & \triangleq & \texttt{C\_SKIP}) \wedge \\
(\texttt{CC (ASSIGN x e)} & \triangleq & \texttt{C\_ASSIGN x (CN e)}) \wedge \\
(\texttt{CC (SEQ c1 c2)} & \triangleq & (\texttt{CC c1}) \texttt{++} (\texttt{CC c2})) \wedge \\
(\texttt{CC (IF b c1 c2)} & \triangleq & \texttt{C\_IF(CB b)(CC c1)(CC c2)} \wedge \\
(\texttt{CC (WHILE b c)} & \triangleq & \texttt{C\_WHILE(CB b)(CC c)})
\end{array}
$$

This relation can be used to 'compile' programs in HOL by a combination of rewriting and unwinding.

5.3 Correctness of Compilation

Verification of the compilation algorithm involves the proof of a correspondence between the semantics of the compiled code and that of the programming language. Having verified the compiler specification, one can be sure that properties shown to hold at the programming language level really do hold when the program is executed on a particular machine.

The precise form of the correspondence depends on the behavioural model used at each level, and on the compilation algorithm; but in general it is some kind of implementation relationship, and one wants to show that the following diagram commutes:



The proof that this relationship holds for all commands is by structural induction.

If one adopts Hoare's approach [Hoa90], representing the machine code semantics in the same way as the high-level programming language, the implementation relationship is a form of refinement and compiler verification becomes an exercise in program transformation.

References

[Bou90] R. J. Boulton. A general-purpose pretty-printer for the HOL system. HOL system documentation, December 1990.

[Gor89] M. J. C. Gordon. Mechanizing programming logic in HOL. Technical Report 145, Computer Laboratory, University of Cambridge, England, 1989.

[Gor91] M. J. C. Gordon. A formal method for hard real-time programming. In *1991 International Workshop on the HOL Theorem Proving System and its Applications*, University of California, Davis, August 1991.

[Hal89] R. W. S. Hale. Programming in temporal logic. Technical Report 173, Computer Laboratory, University of Cambridge, England, 1989.

[Hoa90] C. A. R. Hoare. Refinement algebra proves correctness of compiling specifications. Technical Report PRG-TR-6-90, Oxford University Computing Laboratory, Oxford, England, 1990.

[Joy89] J. J. Joyce. A verified compiler for a verified microprocessor. Technical Report 167, Computer Laboratory, University of Cambridge, England, 1989.

[Mel88] T. F. Melham. Automating recursive type definitions in higher order logic. In G. Birtwhistle and P. A. Subrahmanyam, editors, *Current trends in Hardware verification and automated deduction*. Springer Verlag, 1988.

[Mel91] T. F. Melham. A package for inductive definitions in HOL. In *1991 International Workshop on the HOL Theorem Proving System and its Applications*, University of California, Davis, August 1991.

[Mos86] B. C. Moszkowski. *Executing Temporal Logic Programs*. Cambridge University Press, Cambridge, England, 1986.

[vT90] J. P. van Tassel. A parser generator for the HOL system. HOL system documentation, 1990.