Mechanizing Programming Logics in Higher Order Logic[1]

Michael J.C. Gordon

Computer Laboratory
New Museums Site
Pembroke Street
Cambridge CB2 3QG

SRI International
Suite 23
Millers Yard
Cambridge CB2 1RQ

Abstract

Formal reasoning about computer programs can be based directly on the semantics of the programming language, or done in a special purpose logic like Hoare logic. The advantage of the first approach is that it guarantees that the formal reasoning applies to the language being used (it is well known, for example, that Hoare's assignment axiom fails to hold for most programming languages). The advantage of the second approach is that the proofs can be more direct and natural.

In this paper, an attempt to get the advantages of both approaches is described. The rules of Hoare logic are *mechanically* derived from the semantics of a simple imperative programming language (using the HOL system). These rules form the basis for a simple program verifier in which verification conditions are generated by LCF-style tactics whose validations use the derived Hoare rules. Because Hoare logic is derived, rather than postulated, it is straightforward to mix semantic and axiomatic reasoning. It is also straightforward to combine the constructs of Hoare logic with other application-specific notations. This is briefly illustrated for various logical constructs, including termination statements, VDM-style 'relational' correctness specifications, weakest precondition statements and dynamic logic formulae .

The theory underlying the work presented here is well known. Our contribution is to propose a way of mechanizing this theory in a way that makes certain practical details work out smoothly.

[1]Earlier versions of this paper appear as University of Cambridge Computer Laboratory Technical Report No. 145, and in the book *Current Trends in Hardware Verification and Automated Theorem Proving*, edited by G. Birtwistle and P.A. Subrahmanyam, Springer-Verlag, 1989.

Contents

1 Introduction **3**

2 A simple imperative programming language **4**

3 Hoare logic **4**
3.1 Axioms and rules of Hoare logic 6
3.2 An example proof in Hoare logic 7

4 Higher order logic **10**
4.1 Types 11
4.2 Definitions 12

5 Semantics in logic **12**
5.1 Semantics of commands 14
5.2 Semantics of partial correctness specifications 17

6 Hoare logic as higher order logic **17**
6.1 Derivation of the **skip**-axiom 18
6.2 Derivation of precondition strengthening 19
6.3 Derivation of postcondition weakening 19
6.4 Derivation of the sequencing rule 20
6.5 Derivation of the **if**-rule 20
6.6 Derivation of the **while**-rule 20
6.7 Derivation of the assignment axiom 21

7 Hoare logic in HOL **22**

8 Introduction to tactics and tacticals **29**
8.1 Tactics 29
8.2 Using Tactics to Prove Theorems 30
8.3 Tacticals 31

9 Verification conditions via tactics **32**

10 Termination and total correctness **38**
10.1 Derived rules for total correctness 40
10.2 Tactics for total correctness 40

11 Other programming logic constructs **42**
11.1 VDM-style specifications 43
11.2 Dijkstra's weakest preconditions 45
11.3 Dynamic logic 49

12 Conclusions and future work **50**

13 Acknowledgements **51**

Bibliography **51**

Index **54**

1 Introduction

The work described here is part of a long term project on verifying combined hardware/software systems by mechanized formal proof. The ultimate goal is to develop techniques and tools for modelling and analysing systems like computer controlled chemical plants, fly-by-wire aircraft and microprocessor based car brake controllers. These typically contain both software and hardware and must respond in real time to asynchronous inputs.

This paper concentrates on software verification. A mechanization of Hoare logic is constructed via a representation of it in higher order logic. The main experiment described here is the implementation, in the HOL system [13], of a program verifier for a simple imperative language. This translates correctness questions formulated in Hoare logic [16] to theorem proving problems (called 'verification conditions') in pure predicate logic. This is a standard technique [17], the only novelty in our approach is the way rules of the programming logic are mechanically derived from the semantics of the programming language in which programs are written. This process of generating verification conditions is implemented as a *tactic*[2] in HOL whose validation part uses the derived Hoare rules; it is thus guaranteed to be sound. This way of implementing a verifier ensures that theorems proved with it are logical consequences of the underlying programming language semantics.

Work is already in progress [20] to prove that the semantics used by the verifier (which is described in Section 5 below) is the same as the semantics determined by running the programs on a simple microprocessor. When this proof is completed, it will follow that theorems proved using the verifier are true statements about the actual behaviour of programs when they are executed on hardware.

To explore the flexibility of representing programming logics in higher order logic, three well-known programming logics are examined. Although we do not construct mechanizations of these here, it is shown that doing so would be straightforward.

There is no new mathematical theory in this paper. It is purely a contribution to the methodology of doing proofs mechanically. We hope that our example verifier demonstrates that it is possible to combine both the slickness of a special purpose logic with the rigor of reasoning directly from the reference semantics of a programming language. We hope also that it demonstrates the expressiveness of higher order logic and the flexibility of Milner's LCF approach [11, 30] to interactive proof. The contents of the remaining sections of the paper are as follows:

Section 2 is a description of the example programming language that will be used.

Section 3 is a brief review of Hoare logic. The presentation is adapted from the book *Programming Language Theory and its Implementation* [14].

Section 4 outlines the version of predicate logic used in this paper.

Section 5 gives the semantics of the language described in Section 2 and the semantics of Hoare-style partial correctness specifications.

Section 6 discusses how partial correctness specifications can be regarded as abbreviations for logic formulae. The axioms and rules in Section 3 then become derivable from the semantics in Section 5.

Section 7 is an account of how the derived axioms and rules of Hoare logic can be mechanized in the HOL system.

Section 8 is a brief introduction to tactics and tacticals.

---

[2]Tactics are described in Section 8 below. The idea of implementing verification condition generation with tactics was developed jointly with Tom Melham.

Section 9 shows how verification conditions can be generated using tactics.

Section 10 explains how a weak form of termination statements can be added to Hoare logic and the necessary additional axioms and inference rules derived from suitable definitions. Additional mechanized proof generating tools (in HOL) are also described.

Section 11 explains how VDM-style specifications, weakest preconditions and dynamic logic can be represented in higher order logic.

Section 12 contains concluding remarks and a brief discussion of future work.

2 A simple imperative programming language

The syntax of the little programming language used in this paper is specified by the BNF given below. In this specification, the variable $\mathcal{N}$ ranges over the *numerals* 0, 1, 2 etc, the variable $\mathcal{V}$ ranges over *program variables*[3] X, Y, Z etc, the variables $\mathcal{E}$, $\mathcal{E}_1$, $\mathcal{E}_2$ etc. range over *integer expressions*, the variables $\mathcal{B}$, $\mathcal{B}_1$, $\mathcal{B}_2$ etc. range over *boolean expressions* and the variables $\mathcal{C}$, $\mathcal{C}_1$, $\mathcal{C}_2$ etc. range over *commands*.

$$\mathcal{E} \ ::= \ \mathcal{N} \mid \mathcal{V} \mid \mathcal{E}_1 + \mathcal{E}_2 \mid \mathcal{E}_1 - \mathcal{E}_2 \mid \mathcal{E}_1 \times \mathcal{E}_2 \mid \ldots$$

$$\mathcal{B} \ ::= \ \mathcal{E}_1 = \mathcal{E}_2 \mid \mathcal{E}_1 \leq \mathcal{E}_2 \mid \ldots$$

$$\begin{array}{lll} \mathcal{C} & ::= & \textbf{skip} \\ & \mid & \mathcal{V} := \mathcal{E} \\ & \mid & \mathcal{C}_1 \ ; \ \mathcal{C}_2 \\ & \mid & \textbf{if } \mathcal{B} \textbf{ then } \mathcal{C}_1 \textbf{ else } \mathcal{C}_2 \\ & \mid & \textbf{while } \mathcal{B} \textbf{ do } \mathcal{C} \end{array}$$

Note that the BNF syntax above is ambiguous: it does not specify, for example, whether **if** $\mathcal{B}$ **then** $\mathcal{C}_1$ **else** $\mathcal{C}_2$; $\mathcal{C}_3$ means (**if** $\mathcal{B}$ **then** $\mathcal{C}_1$ **else** $\mathcal{C}_2$) ; $\mathcal{C}_3$ or **if** $\mathcal{B}$ **then** $\mathcal{C}_1$ **else** ($\mathcal{C}_2$; $\mathcal{C}_3$). We will clarify, whenever necessary, using brackets. Here, for example, is a command $\mathcal{C}$ to compute the quotient Q and remainder R that results from dividing Y into X.

$$\left.\begin{array}{l} R := X; \\ Q := 0; \\ \textbf{while } Y \leq R \textbf{ do} \\ \quad (R := R - Y;\ Q := Q + 1) \end{array}\right\} \mathcal{C}$$

3 Hoare logic

In a seminal paper, C.A.R. Hoare [16] introduced the notation $\{\mathcal{P}\}\ \mathcal{C}\ \{\mathcal{Q}\}$ for specifying what a program does[4]. In this notation, $\mathcal{C}$ is a program from the programming

[3] To distinguish program variables from logical variables, the convention is adopted here that the former are upper case and the latter are lower case. The need for such a convention is explained in Section 5.

[4] Actually, Hoare's original notation was $\mathcal{P}\ \{\mathcal{C}\}\ \mathcal{Q}$ not $\{\mathcal{P}\}\ \mathcal{C}\ \{\mathcal{Q}\}$, but the latter form is now more widely used. Note that some authors (e.g. [15]) use $\{\mathcal{P}\}\ \mathcal{C}\ \{\mathcal{Q}\}$ for total correctness rather than partial correctness.

language whose programs are being specified (the language in Section 2 in our case); and $\mathcal{P}$ and $\mathcal{Q}$ are conditions on the program variables used in $\mathcal{C}$.

The semantics of $\{\mathcal{P}\}\ \mathcal{C}\ \{\mathcal{Q}\}$ is now described informally. A formal semantics in higher order logic is given in Section 5 below.

The effect of executing a command $\mathcal{C}$ is to change the *state*, where a state is simply an assignment of values to program variables. If $\mathcal{F}[X_1, \ldots, X_n]$ is a formula containing free[5] program variables $X_1, \ldots, X_n$, then we say $\mathcal{F}[X_1, \ldots, X_n]$ is true in a state s if $\mathcal{F}[X_1, \ldots, X_n]$ is true when $X_1, \ldots, X_n$ have the values assigned to them by s. The $\mathcal{P}$ and $\mathcal{Q}$ in $\{\mathcal{P}\}\ \mathcal{C}\ \{\mathcal{Q}\}$ are logic formulae containing the program variables used in $\mathcal{C}$ (and maybe other variables also).

$\{\mathcal{P}\}\ \mathcal{C}\ \{\mathcal{Q}\}$ is said to be true, if whenever $\mathcal{C}$ is executed in a state in which $\mathcal{P}$ is true, and if the execution of $\mathcal{C}$ terminates, then $\mathcal{Q}$ is true in the state in which $\mathcal{C}$'s execution terminates.

Example

$\{X = 1\}\ X := X + 1\ \{X = 2\}$. Here $\mathcal{P}$ is the condition that the value of X is 1, $\mathcal{Q}$ is the condition that the value of X is 2 and $\mathcal{C}$ is the assignment command $X := X + 1$ (i.e. 'X becomes $X + 1$'). □

An expression $\{\mathcal{P}\}\ \mathcal{C}\ \{\mathcal{Q}\}$ is called a *partial correctness specification*; $\mathcal{P}$ is called its *precondition* and $\mathcal{Q}$ its *postcondition*.

These specifications are 'partial' because for $\{\mathcal{P}\}\ \mathcal{C}\ \{\mathcal{Q}\}$ to be true it is *not* necessary for the execution of $\mathcal{C}$ to terminate when started in a state satisfying $\mathcal{P}$. It is only required that *if* the execution terminates, *then* $\mathcal{Q}$ holds.

Example

$$
\begin{array}{l}
\{\mathsf{T}\} \\
\quad \left.\begin{array}{l}
R := X; \\
Q := 0; \\
\textbf{while}\ Y \leq R\ \textbf{do} \\
\quad (R := R - Y;\ Q := Q + 1)
\end{array}\right\}\mathcal{C} \\
\{X = R + (Y \times Q)\ \wedge\ R < Y\}
\end{array}
$$

This is $\{\mathsf{T}\}\ \mathcal{C}\ \{X = R + (Y \times Q)\ \wedge\ R < Y\}$ where $\mathcal{C}$ is the command indicated by the braces above. The formula T making up the precondition is the universally true formula; the symbol $\wedge$ denotes logical conjunction (i.e. 'and'). The specification is true if whenever the execution of $\mathcal{C}$ halts, then Q is the quotient and R is the remainder that results from dividing Y into X. It is true, even if X is initially negative. □

A stronger kind of specification is a *total correctness specification*. There is no standard notation for these. We will use $[\mathcal{P}]\ \mathcal{C}\ [\mathcal{Q}]$.

A total correctness specification $[\mathcal{P}]\ \mathcal{C}\ [\mathcal{Q}]$ is true if and only if the following conditions apply:

(i) Whenever $\mathcal{C}$ is executed in a state satisfying $\mathcal{P}$, then the execution of $\mathcal{C}$ terminates.

(ii) After termination $\mathcal{Q}$ holds.

The relationship between partial and total correctness can be informally expressed by the equation:

[5] A *free* occurrence of a variable is one that is not bound by $\forall$, $\exists$ or λ etc.

$$\text{Total correctness} = \text{Termination} + \text{Partial correctness.}$$

It is usually easier to prove total correctness by establishing termination and partial correctness separately. We show how this separation of concerns is supported by our program verifier in Section 10.

3.1 Axioms and rules of Hoare logic

In his 1969 paper, Hoare gave a deductive system for partial correctness specifications. The axioms and rules that follow are minor variants of these[6]. We write $\vdash \{\mathcal{P}\}\ \mathcal{C}\ \{\mathcal{Q}\}$ if $\{\mathcal{P}\}\ \mathcal{C}\ \{\mathcal{Q}\}$ is either an instance of one of the axiom schemes A1 or A2 below, or can be deduced by a sequence of applications of the rules R1, R2, R3, R4 or R5 below from such instances. We write $\vdash \mathcal{P}$, where $\mathcal{P}$ is a formula of predicate logic, if $\mathcal{P}$ can be deduced from the laws of logic and arithmetic. We shall not give an axiom system for either predicate logic or arithmetic here. In the little program verifier that we describe in Section 9, such logical and arithmetical theorems are deduced using the existing proof infrastructure of the HOL system [13]. The goal of the work described in this paper is to gracefully embed Hoare logic in higher order logic, so that the HOL system can also be used to verify progams via the axioms and rules below.

If $\vdash \mathcal{P}$, where $\mathcal{P}$ is a formula of predicate calculus or arithmetic, then we say '$\vdash \mathcal{P}$ is a theorem of pure logic'; if $\vdash \{\mathcal{P}\}\ \mathcal{C}\ \{\mathcal{Q}\}$ we say '$\vdash \{\mathcal{P}\}\ \mathcal{C}\ \{\mathcal{Q}\}$ is a theorem of Hoare logic'.

A1: the skip-axiom. For any formula $\mathcal{P}$:

$$\vdash \{\mathcal{P}\}\ \mathbf{skip}\ \{\mathcal{P}\}$$

A2: the assignment-axiom. For any formula $\mathcal{P}$, program variable $\mathcal{V}$ and integer expression $\mathcal{E}$:

$$\vdash \{\mathcal{P}[\mathcal{E}/\mathcal{V}]\}\ \mathcal{V} := \mathcal{E}\ \{\mathcal{P}\}$$

where $\mathcal{P}[\mathcal{E}/\mathcal{V}]$ denotes the result of substituting $\mathcal{E}$ for all free occurrences of $\mathcal{V}$ in $\mathcal{P}$ (and free variables are renamed, if necessary, to avoid capture).

Rules R1 to R5 below are stated in standard notation: the hypotheses of the rule above a horizontal line and the conclusion below it. For example, R1 states that if $\vdash \mathcal{P}' \Rightarrow \mathcal{P}$ is a theorem of pure logic and $\vdash \{\mathcal{P}\}\ \mathcal{C}\ \{\mathcal{Q}\}$ is a theorem of Hoare logic, then $\vdash \{\mathcal{P}'\}\ \mathcal{C}\ \{\mathcal{Q}\}$ can be deduced by R1.

R1: the rule of precondition strengthening. For any formulae $\mathcal{P}$, $\mathcal{P}'$ and $\mathcal{Q}$, and command $\mathcal{C}$:

$$\frac{\vdash \mathcal{P}' \Rightarrow \mathcal{P} \qquad \vdash \{\mathcal{P}\}\ \mathcal{C}\ \{\mathcal{Q}\}}{\vdash \{\mathcal{P}'\}\ \mathcal{C}\ \{\mathcal{Q}\}}$$

R2: the rule of postcondition weakening. For any formulae $\mathcal{P}$, $\mathcal{Q}$ and $\mathcal{Q}'$, and command $\mathcal{C}$:

[6]The presentation of Hoare logic in this paper is based on the one in *Programming Language Theory and its Implementation* [14].

$$\frac{\vdash\ \{\mathcal{P}\}\ \mathcal{C}\ \{\mathcal{Q}\} \qquad \vdash\ \mathcal{Q}\ \Rightarrow\ \mathcal{Q}'}{\vdash\ \{\mathcal{P}\}\ \mathcal{C}\ \{\mathcal{Q}'\}}$$

Notice that in R1 and R2, one hypothesis is a theorem of ordinary logic whereas the other hypothesis is a theorem of Hoare logic. This shows that proofs in Hoare logic may require subproofs in pure logic; more will be said about the implications of this later.

R3: the sequencing rule. For any formulae $\mathcal{P}$, $\mathcal{Q}$ and $\mathcal{R}$, and commands $\mathcal{C}_1$ and $\mathcal{C}_2$:

$$\frac{\vdash\ \{\mathcal{P}\}\ \mathcal{C}_1\ \{\mathcal{Q}\} \qquad \vdash\ \{\mathcal{Q}\}\ \mathcal{C}_2\ \{\mathcal{R}\}}{\vdash\ \{\mathcal{P}\}\ \mathcal{C}_1;\ \mathcal{C}_2\ \{\mathcal{R}\}}$$

R4: the if-rule. For any formulae $\mathcal{P}$, $\mathcal{Q}$ and $\mathcal{B}$, and commands $\mathcal{C}_1$ and $\mathcal{C}_2$:

$$\frac{\vdash\ \{\mathcal{P}\ \wedge\ \mathcal{B}\}\ \mathcal{C}_1\ \{\mathcal{Q}\} \qquad \vdash\ \{\mathcal{P}\ \wedge\ \neg\mathcal{B}\}\ \mathcal{C}_2\ \{\mathcal{Q}\}}{\vdash\ \{\mathcal{P}\}\ \textbf{if}\ \mathcal{B}\ \textbf{then}\ \mathcal{C}_1\ \textbf{else}\ \mathcal{C}_2\ \{\mathcal{Q}\}}$$

Notice that in this rule (and also in R5 below) it is assumed that $\mathcal{B}$ is both a boolean expression of the programming language and a formula of predicate logic. We shall only assume that the boolean expressions of the language are a *subset* of those in predicate logic. This assumption is reasonable since we are the designers of our example language and can design the language so that it is true; it would not be reasonable if we were claiming to provide a logic for reasoning about an existing language like Pascal. One consequence of this assumption is that the semantics of boolean expressions must be the usual logical semantics. We could not, for example, have 'sequential' boolean operators in which the boolean expression $\mathsf{T}\ \vee\ (1/0 = 0)$ evaluates to T, but $(1/0 = 0)\ \vee\ \mathsf{T}$ causes an error (due to division by 0).

R5: the while-rule. For any formulae $\mathcal{P}$ and $\mathcal{B}$, and command $\mathcal{C}$:

$$\frac{\vdash\ \{\mathcal{P}\ \wedge\ \mathcal{B}\}\ \mathcal{C}\ \{\mathcal{P}\}}{\vdash\ \{\mathcal{P}\}\ \textbf{while}\ \mathcal{B}\ \textbf{do}\ \mathcal{C}\ \{\mathcal{P}\ \wedge\ \neg\mathcal{B}\}}$$

A formula $\mathcal{P}$ such that $\vdash\ \{\mathcal{P}\ \wedge\ \mathcal{B}\}\ \mathcal{C}\ \{\mathcal{P}\}$ is called an *invariant* of $\mathcal{C}$ for $\mathcal{B}$.

3.2 An example proof in Hoare logic

The simple proof that follows will be used later to illustrate the mechanization of Hoare logic in the HOL system.

By the assignment axiom:

$$\text{Th1}:\quad \vdash\ \{X = X + (Y \times 0)\}\ R := X\ \{X = R + (Y \times 0)\}$$

$$\text{Th2}:\quad \vdash\ \{X = R + (Y \times 0)\}\ Q := 0\ \{X = R + (Y \times Q)\}$$

Hence by the sequencing rule:

$$\text{Th3}:\quad \vdash\ \{X = X + (Y \times 0)\}\ R := X;\ Q := 0\ \{X = R + (Y \times Q)\}$$

By a similar argument consisting of two instances of the assignment axiom followed by a use of the sequencing rule:

$$\begin{array}{lll}\text{Th4}: & \vdash & \{X = (R-Y)+(Y\times(Q+1))\} \\ & & \quad R := R-Y \\ & & \{X = R+(Y\times(Q+1))\}\end{array}$$

$$\text{Th5}: \quad \vdash \ \{X = R+(Y\times(Q+1))\}\ Q := Q+1\ \{X = R+(Y\times Q)\}$$

$$\begin{array}{lll}\text{Th6} & \vdash & \{X = (R-Y)+(Y\times(Q+1))\} \\ & & \quad R := R-Y;\ Q := Q+1 \\ & & \{X = R+(Y\times Q)\}\end{array}$$

The following is a trivial theorem of arithmetic:

$$\text{Th7}: \quad \vdash \ (X = R+(Y\times Q)) \ \wedge\ Y \leq R \ \Rightarrow\ (X = (R-Y)+(Y\times(Q+1)))$$

hence by precondition strengthening applied to Th7 and Th6:

$$\begin{array}{lll}\text{Th8}: & \vdash & \{(X = R+(Y\times Q)) \ \wedge\ Y \leq R\} \\ & & \quad R := R-Y;\ Q := Q+1 \\ & & \{X = R+(Y\times Q)\}\end{array}$$

and so by the **while**-rule

$$\begin{array}{lll}\text{Th9}: & \vdash & \{X = R+(Y\times Q)\} \\ & & \quad \textbf{while}\ Y \leq R\ \textbf{do}\ (R := R-Y;\ Q := Q+1) \\ & & \{(X = R+(Y\times Q)) \ \wedge\ \neg Y \leq R\}\end{array}$$

By the sequencing rule applied to Th3 and Th9

$$\begin{array}{lll}\text{Th10}: & \vdash & \{X = X+(Y\times 0)\} \\ & & \quad R := X; \\ & & \quad Q := 0; \\ & & \quad \textbf{while}\ Y \leq R\ \textbf{do}\ (R := R-Y;\ Q := Q+1) \\ & & \{(X = R+(Y\times Q)) \ \wedge\ \neg Y \leq R\}\end{array}$$

The next two theorems are trivial facts of arithmetic:

$$\text{Th11}: \quad \vdash \ \mathsf{T} \ \Rightarrow\ X = X+(Y\times 0)$$

$$\text{Th12}: \quad \vdash \ (X = R+(Y\times Q)) \ \wedge\ \neg Y \leq R \ \Rightarrow\ (X = R+(Y\times Q)) \ \wedge\ R < Y$$

Finally, combining the last three theorems using precondition strengthening and postcondition weakening:

$$\begin{array}{lll}\text{Th13}: & \vdash & \{\mathsf{T}\} \\ & & \quad R := X; \\ & & \quad Q := 0; \\ & & \quad \textbf{while}\ Y \leq R\ \textbf{do}\ (R := R-Y;\ Q := Q+1) \\ & & \{(X = R+(Y\times Q)) \ \wedge\ R < Y\}\end{array}$$

In the example just given, it was shown how to prove $\{\mathcal{P}\}\mathcal{C}\{\mathcal{Q}\}$ by proving properties of the components of $\mathcal{C}$ and then putting these together (with the appropriate

proof rules) to get the desired property of $\mathcal{C}$ itself. For example, Th3 and Th6 both had the form $\vdash \{\mathcal{P}\}\mathcal{C}_1;\mathcal{C}_2\{\mathcal{Q}\}$ and to prove them we first proved $\vdash \{\mathcal{P}\}\mathcal{C}_1\{\mathcal{R}\}$ and $\vdash \{\mathcal{R}\}\mathcal{C}_2\{\mathcal{Q}\}$ (for suitable $\mathcal{R}$), and then deduced $\vdash \{\mathcal{P}\}\mathcal{C}_1;\mathcal{C}_2\{\mathcal{Q}\}$ by the sequencing rule.

This process is called *forward proof*, because one moves forward from axioms via rules to conclusions. In practice, it is much more natural to work backwards: starting from the goal of showing $\{\mathcal{P}\}\mathcal{C}\{\mathcal{Q}\}$, one generates subgoals, subsubgoals etc. until the problem is solved. For example, suppose one wants to show:

$$\{X = x \ \wedge \ Y = y\}\ R := X;\ X := Y;\ Y := R\ \{Y = x \ \wedge \ X = y\}$$

then by the assignment axiom and sequencing rule it is sufficient to show the subgoal

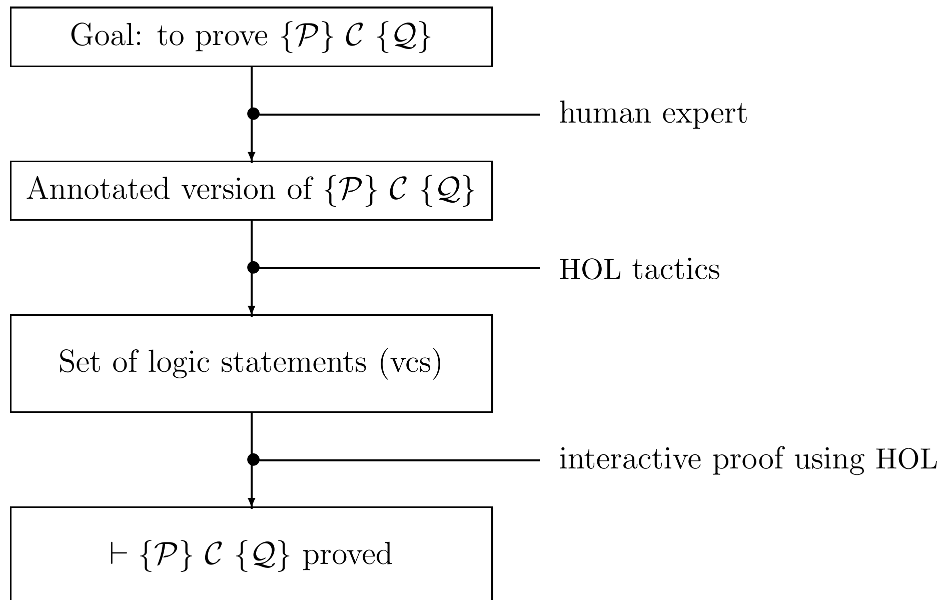
$$\{X = x \ \wedge \ Y = y\}\ R := X;\ X := Y\ \{R = x \ \wedge \ X = y\}$$

(because $\vdash \ \{R = x \ \wedge \ X = y\}\ Y := R\ \{Y = x \ \wedge \ X = y\}$). By a similar argument this subgoal can be reduced to

$$\{X = x \ \wedge \ Y = y\}\ R{:=}X\ \{R = x \ \wedge \ Y = y\}$$

which clearly follows from the assignment axiom.

In Section 9 we describe how LCF style tactics (which are described in Section 8) can be used to implement a *goal oriented* method of proof based on verification conditions [17]. The user supplies a partial correctness specification annotated with mathematical statements describing relationships between variables (e.g. **while** invariants). HOL tactics can then be used to generate a set of purely mathematical statements, called *verification conditions* (or vcs). The validation of the vc-generating tactic (see Section 8) ensures that if the verification conditions are provable, then the original specification can be deduced from the axioms and rules of Floyd-Hoare logic. The following diagram (adapted from [14]) gives an overview of this approach.



The next section explains the version of predicate calculus underlying the HOL system; the section after that is a quick introduction to Milner's ideas on tactics and tacticals.

4 Higher order logic

The table below shows the logical notations used in this paper.

Predicate calculus notation	
Notation	*meaning*
T	truth
F	falsity
$P(x)$ (or $P\ x$)	x has property P
$\neg t$	not t
$t_1 \vee t_2$	t_1 or t_2
$t_1 \wedge t_2$	t_1 and t_2
$t_1 \Rightarrow t_2$	t_1 implies t_2
$t_1 \equiv t_2$	t_1 if and only if t_2
$t_1 = t_2$	t_1 equals t_2
$\forall x.\ t[x]$	for all x it is the case that $t[x]$
$\exists x.\ t[x]$	for some x it is the case that $t[x]$
$(t \rightarrow t_1 \mid t_2)$	if t is true then t_1 else t_2

Higher order logic extends first-order logic by allowing higher order variables (i.e. variables whose values are functions) and higher order functions (i.e. functions whose arguments and/or results are other functions). For example[7], partial correctness specifications can be represented by defining a predicate (i.e. a function whose result is a truth value) Spec by:

$$\mathsf{Spec}(p, c, q) \ = \ \forall s_1\ s_2.\ p\ s_1\ \wedge\ c(s_1, s_2)\ \Rightarrow\ q\ s_2$$

Spec is a predicate on triples (p, c, q) where p and q are unary predicates and c is a binary predicate. To represent command sequencing we can define a constant Seq by:

$$\mathsf{Seq}(c_1, c_2)(s_1, s_2) \ = \ \exists s.\ c_1(s_1, s)\ \wedge\ c_2(s, s_2)$$

The sequencing rule in Hoare logic (which was explained in Section 3) can be stated directly in higher order logic as:

$$\vdash\ \forall p\ q\ r\ c_1\ c_2.\ \mathsf{Spec}(p, c_1, q)\ \wedge\ \mathsf{Spec}(q, c_2, r)\ \Rightarrow\ \mathsf{Spec}(p, \mathsf{Seq}(c_1, c_2), r)$$

These examples make essential use of higher order variables; they can't be expressed in first-order logic.

The version of higher order logic that we use[8] has function-denoting terms called λ-expressions. These have the form $\lambda x.\ t$ where x is a variable and t is an expression. Such a λ-term denotes the function $a \mapsto t[a/x]$ where $t[a/x]$ is the result of substituting a for x in t. For example, $\lambda x.\ x{+}3$ denotes the function $a \mapsto a{+}3$

[7] The examples here are explained in more detail in Sections 5.1 and 5.2 below.

[8] The version of higher order logic used in this paper is a slight extension of a system invented by Church [5]. Church's system is sometimes called 'simple type theory'; an introductory textbook on this has recently been written by Andrews [1].

which adds 3 to its argument. The simplification of $(\lambda x.\ t)t'$ to $t[t'/x]$ is called β-reduction. λ-expressions of the form $\lambda(x_1,\ldots,x_n).\ t$ will also be used; these denote functions defined on n-tuples. For example, $\lambda(m,n).\ m+n$ denotes the function $(m,n)\mapsto m+n$.

To save writing brackets, function applications can be written as $f\ x$ instead of $f(x)$. More generally, we adopt the standard convention that $t_1\ t_2\ t_3\ \cdots\ t_n$ abbreviates $(\ \cdots\ ((t_1\ t_2)\ t_3)\ \cdots\ t_n)$ *i.e.* application associates to the left.

The notation $\lambda x_1\ x_2\ \cdots\ x_n.\ t$ abbreviates $\lambda x_1.\ \lambda x_2.\ \cdots\ \lambda x_n.\ t$. The scope of a λ extends as far to the right as possible. Thus, for example, $\lambda b.\ b = \lambda x.\ \mathsf{T}$ means $\lambda b.\ (b = (\lambda x.\ \mathsf{T}))$ not $(\lambda b.\ b) = (\lambda x.\ \mathsf{T})$.

4.1 Types

Higher order variables can be used to formulate Russell's paradox: define the predicates P and Ω by:

$$\begin{aligned}\mathsf{P}\ x &= \neg\ (x\ x)\\ \Omega &= \mathsf{P}\ \mathsf{P}\end{aligned}$$

Then it immediately follows that $\Omega = \neg\ \Omega$. Russell invented his theory of types to prevent such inconsistencies. Church simplified Russell's idea; HOL uses a slight extension of Church's type system[9].

Types are expressions that denote sets of values, they are either *atomic* or *compound*. Examples of atomic types are:

$$\mathit{bool},\qquad \mathit{num},\qquad \mathit{real},\qquad \mathit{string}$$

these denote the sets of booleans, natural numbers, real numbers and character strings respectively. Compound types are built from atomic types (or other compound types) using *type operators*. For example, if σ, σ_1 and σ_2 are types then so are:

$$\sigma\ \mathit{list},\qquad \sigma_1\times\sigma_2,\qquad \sigma_1{\to}\sigma_2$$

where *list* is a postfixed unary type operator and $\to$ and $\times$ are infixed binary type operators. The type σ *list* denotes the set of lists of values of elements from the set denoted by σ; the type $\sigma_1\times\sigma_2$ denotes the set of pairs (x_1,x_2) where x_1 is in the set denoted by σ_1 and x_2 is in the set denoted by σ_2; the type $\sigma_1{\to}\sigma_2$ denotes the set of total functions with domain denoted by σ_1 and range denoted by σ_2. Lower case *slanted* identifiers will be used for particular types, and greek letters (mostly σ) to range over arbitrary types.

Terms of higher order logic must be *well-typed* in the sense that each subterm can be assigned a type 'in a consistent way'. More precisely, it must be possible to assign a type to each subterm such that both 1 and 2 below hold.

1. For every subterm of the form $t_1\ t_2$ there are types σ and σ' such that:
   (a) t_1 is assigned $\sigma'{\to}\sigma$
   (b) t_2 is assigned σ'
   (c) $t_1\ t_2$ is assigned the type σ.
2. Every subterm of the form $\lambda x.\ t$ is assigned a type $\sigma_1{\to}\sigma_2$ where:

[9] The type system of the HOL logic is the type system used in LCF's logic PPλ [11, 30].

(a) x is assigned σ_1

(b) t is assigned σ_2.

Variables with the same name can be assigned different types, but then they are regarded as different variables.

Writing $t{:}\sigma$ indicates that a term t has type σ. Thus $x{:}\sigma_1$ is the same variable as $x{:}\sigma_2$ if and only if $\sigma_1 = \sigma_2$. In Church's original notation (which is also used by Andrews) $t{:}\sigma$ would be written t_σ.

In some formulations of higher-order logic, the types of variables have to be written down explicitly. For example, $\lambda x.\ \mathsf{cos}(\mathsf{sin}(x))$ would not be allowed in Church's system, instead one would have to write:

$$\lambda x_{real}.\ \mathsf{cos}_{real\to real}(\mathsf{sin}_{real\to real}(x_{real}))$$

We allow the types of variables to be omitted if they can be inferred from the context. There is an algorithm, due to Robin Milner [27], for doing such type inference.

We adopt the standard conventions that $\sigma_1{\to}\sigma_2{\to}\sigma_3{\to}\ \cdots\ \sigma_n{\to}\sigma$ is an abbreviation for $\sigma_1{\to}(\sigma_2{\to}(\sigma_3{\to}\ \cdots\ (\sigma_n{\to}\sigma)\ \cdots))$ *i.e.* $\to$ associates to the right. This convention blends well with the left associativity of function application, because if f has type $\sigma_1{\to}\ \cdots\ \sigma_n{\to}\sigma$ and $t_1, \ldots, t_n$ have types $\sigma_1, \ldots, \sigma_n$ respectively, then $f\ t_1\ \cdots\ t_n$ is a well-typed term of type σ. We also assume $\times$ is more tightly binding than $\to$; for example, *state* $\times$ *state*$\to$*bool* means (*state* $\times$ *state*)$\to$*bool*.

4.2 Definitions

In addition to the 'built-in' constants like $\wedge$, $\vee$, $\neg$ etc, new constants can be introduced by *definitions*. The definition of a constant, c say, is an axiom of the form:

$$\vdash\ \mathsf{c}\ =\ t$$

where t is a closed[10] term. For example, the definition of Seq given in Section 5 below is:

$$\vdash\ \mathsf{Seq}\ =\ \lambda(C_1, C_2).\ \lambda(s_1, s_2).\ \exists s.\ C_1(s_1, s)\ \wedge\ C_2(s, s_2)$$

which is logically equivalent to:

$$\vdash\ \mathsf{Seq}(C_1, C_2)(s_1, s_2)\ =\ \exists s.\ C_1(s_1, s)\ \wedge\ C_2(s, s_2)$$

New types can also be defined as names for subsets of existing types; see Melham's paper for details [25]. A particular collection of constants, types and definitions is called a *theory*. Theories can be hierarchically structured and stored on disk [13].

5 Semantics in logic

The traditional denotation of a command $\mathcal{C}$ is a function, $\mathsf{Meaning}(\mathcal{C})$ say, from machine states to machine states. The idea is:

$$\mathsf{Meaning}(\mathcal{C})(s)\ =\ \text{`the state resulting from executing } \mathcal{C} \text{ in state } s\text{'}$$

[10] A term is closed if it has no free variables; i.e. all variables are bound by $\forall$, $\exists$ or λ.

Since **while**-commands need not terminate, the functions denoted by commands will be *partial*. For example, for any state s and command $\mathcal{C}$

$$\mathsf{Meaning}(\textbf{while}\ \mathsf{T}\ \textbf{do}\ \mathcal{C})(s)$$

will be undefined. Since functions in conventional predicate calculus are total,[11] we cannot use them as command denotations. Instead we will take the meaning of commands to be predicates on pairs of states (s_1, s_2); the idea being that if $\mathcal{C}$ denotes c then:

$$c(s_1, s_2) \quad \equiv \quad (\mathsf{Meaning}(\mathcal{C})(s_1) \ = \ s_2)$$

i.e.

$$c(s_1, s_2) = \begin{cases} \mathsf{T} & \text{if executing } \mathcal{C} \text{ in state } s_1 \text{ results in state } s_2 \\ \\ \mathsf{F} & \text{otherwise} \end{cases}$$

If c_{while} is the predicate denoted by **while** T **do** $\mathcal{C}$, we will simply have:

$$\forall s_1\ s_2.\ \mathsf{c}_{while}(s_1, s_2) = \mathsf{F}$$

Formally, the type *state* of states that we use is defined by:

$$\mathit{state} \quad = \quad \mathit{string} {\rightarrow} \mathit{num}$$

The notation 'XYZ' will be used for the string consisting of the three characters X, Y and Z; thus 'XYZ' : *string*. A state s in which the strings 'X', 'Y' and 'Z' are bound to 1, 2 and 3 respectively, and all other strings are bound to 0, is defined by:

$$s \quad = \quad \lambda x.\ (x = \text{`}X\text{`} \rightarrow 1 \mid (x = \text{`}Y\text{`} \rightarrow 2 \mid (x = \text{`}Z\text{`} \rightarrow 3 \mid 0)))$$

If e, b and c are the denotations of $\mathcal{E}$, $\mathcal{B}$ and $\mathcal{C}$ respectively, then:

$$\begin{array}{lcl} e & : & \mathit{state} {\rightarrow} \mathit{num} \\ b & : & \mathit{state} {\rightarrow} \mathit{bool} \\ c & : & \mathit{state} \times \mathit{state} {\rightarrow} \mathit{bool} \end{array}$$

For example, the denotation of $X + 1$ would be $\lambda s.\ s\text{`}X\text{`} + 1$ and the denotation of $(X + Y) > 10$ would be $\lambda s.\ (s\text{`}X\text{`} + s\text{`}Y\text{`}) > 10$.

It is convenient to introduce the notations $[\![\mathcal{E}]\!]$ and $[\![\mathcal{B}]\!]$ for the logic terms representing the denotations of $\mathcal{E}$ and $\mathcal{B}$. For example:

$$\begin{array}{lcl} [\![X + 1]\!] & = & \lambda s.\ s\text{`}X\text{`} + 1 \\ [\![(X + Y) > 10]\!] & = & \lambda s.\ (s\text{`}X\text{`} + s\text{`}Y\text{`}) > 10 \end{array}$$

Note that $[\![\mathcal{E}]\!]$ and $[\![\mathcal{B}]\!]$ are *terms*, i.e. syntactic objects. In traditional denotational semantics, the meanings of expressions are represented by abstract mathematical entities, like functions. Such abstract entities are the 'meanings' of $[\![\mathcal{E}]\!]$ and $[\![\mathcal{B}]\!]$; but

[11] There are versions of predicate calculus than can handle partial functions; for example, the Scott/Milner 'Logic of Computable Functions' (LCF) [30] and the Scott/Fourman formulation of intuitionisitic higher order logic [24]. The HOL system used in this paper is actually a version of Milner's proof assistant for LCF [11] which has been modified to support higher order logic. Although programming language semantics are particularly easy to represent in LCF, other kinds of semantics (e.g. the behaviour of hardware) are more straightforward in classical logic.

because we are concerned with mechanical reasoning, we work with the terms and formulae themselves, rather than with the intangible abstract entities they denote.

Sometimes it is necessary for pre and postconditions to contain logical variables that are not program variables. An example is:

$$\{X = x \ \wedge\ Y = y\}\ Z \ \texttt{:=}\ X\texttt{;}\ X \ \texttt{:=}\ Y\texttt{;}\ Y \ \texttt{:=}\ Z\ \{X = y \ \wedge\ Y = x\}$$

Here x and y are logical variables whereas X and Y (and Z) are program variables. The formulae representing the correct semantics of the pre and post conditions of this specification are:

$$\begin{array}{lcl} [\![X = x \ \wedge\ Y = y]\!] & = & \lambda s.\ s`X` = x \ \wedge\ s`Y` = y \\ [\![X = y \ \wedge\ Y = x]\!] & = & \lambda s.\ s`X` = y \ \wedge\ s`Y` = x \end{array}$$

The convention adopted in this paper is that upper case variables are program variables and lower case variables are logical variables (as in the example just given). Logical variables occurring in pre and post conditions are sometimes called *ghost variables* or *auxiliary variables*. In our little programming language the only data type is numbers, hence program variables will have type *num*. The definition of $[\![\cdots]\!]$ can now be stated more precisely: if $\mathcal{T}[X_1, \ldots, X_n]$ is a term of higher order logic whose upper case free variables of type *num* are X_1, ... , X_n then

$$[\![\mathcal{T}[X_1, \ldots, X_n]]\!] \quad = \quad \lambda s.\ \mathcal{T}[s`X_1`, \ldots, s`X_n`]$$

In other words if $\mathcal{T}$ is a term of type σ then the term $[\![\mathcal{T}]\!]$ of type $state{\rightarrow}\sigma$ is obtained as follows:

(i) Each free upper case variable $\mathcal{V}$ of type *num* is replaced by the term $s`\mathcal{V}`$, where s is a variable of type *state* not occurring in $\mathcal{P}$.

(ii) The result of (i) is prefixed by '$\lambda s.$'.

5.1 Semantics of commands

To represent the semantics of our little programming language, predicates in higher order logic that correspond to the five kinds of commands are defined. For each command $\mathcal{C}$, a term $[\![\mathcal{C}]\!]$ of type $state \times state{\rightarrow}bool$ is defined as follows:

1. $[\![\textbf{skip}]\!] \ = \ \mathsf{Skip}$

   where the constant Skip is defined by:

$$\mathsf{Skip}(s_1, s_2) \ = \ (s_1 = s_2)$$

2. $[\![\mathcal{V} \ \texttt{:=}\ \mathcal{E}]\!] \ = \ \mathsf{Assign}(`\mathcal{V}`, [\![\mathcal{E}]\!])$

   where the constant Assign is defined by:

$$\mathsf{Assign}(v, e)(s_1, s_2) \quad = \quad (s_2 = \mathsf{Bnd}(e, v, s_1))$$

   where:

$$\mathsf{Bnd}(e, v, s) \ = \ \lambda x.\ (x = v \rightarrow e\ s \mid s\ x)$$

3. $[\![\mathcal{C}_1 \,;\, \mathcal{C}_2]\!] \;=\; \mathsf{Seq}([\![\mathcal{C}_1]\!], [\![\mathcal{C}_2]\!])$

   where the constant Seq is defined by:

$$\mathsf{Seq}(c_1, c_2)(s_1, s_2) \;=\; \exists s.\; c_1(s_1, s) \;\wedge\; c_2(s, s_2)$$

4. $[\![\textbf{if } \mathcal{B} \textbf{ then } \mathcal{C}_1 \textbf{ else } \mathcal{C}_2]\!] \;=\; \mathsf{If}([\![\mathcal{B}]\!], [\![\mathcal{C}_1]\!], [\![\mathcal{C}_2]\!])$

   where the constant If is defined by:

$$\mathsf{If}(b, c_1, c_2)(s_1, s_2) \;=\; (b\ s_1 \rightarrow c_1(s_1, s_2) \mid c_2(s_1, s_2))$$

5. $[\![\textbf{while } \mathcal{B} \textbf{ do } \mathcal{C}]\!] \;=\; \mathsf{While}([\![\mathcal{B}]\!], [\![\mathcal{C}]\!])$

   where the constant While is defined by:

$$\mathsf{While}(b, c)(s_1, s_2) \;=\; \exists n.\; \mathsf{Iter}(n)(b, c)(s_1, s_2)$$

   where $\mathsf{Iter}(n)$ is defined by primitive recursion as follows:

$$\begin{array}{lcl} \mathsf{Iter}(0)(b, c)(s_1, s_2) & = & \mathsf{F} \\ \mathsf{Iter}(n{+}1)(b, c)(s_1, s_2) & = & \mathsf{If}(b, \mathsf{Seq}(c, \mathsf{Iter}(n)(b, c)), \mathsf{Skip})(s_1, s_2) \end{array}$$

Example

R := X ;
Q := 0;
while $Y \leq R$
  do (R := $R - Y$; Q := $Q + 1$)

denotes:

$$
\begin{array}{l}
\mathsf{Seq} \\
\quad (\mathsf{Assign}(\text{`}R\text{`}, [\![X]\!]), \\
\quad\quad \mathsf{Seq} \\
\quad\quad\quad (\mathsf{Assign}(\text{`}Q\text{`}, [\![0]\!]), \\
\quad\quad\quad\quad \mathsf{While} \\
\quad\quad\quad\quad\quad ([\![Y \leq R]\!], \\
\quad\quad\quad\quad\quad\quad \mathsf{Seq} \\
\quad\quad\quad\quad\quad\quad\quad (\mathsf{Assign}(\text{`}R\text{`}, [\![R - Y]\!], \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathsf{Assign}(\text{`}Q\text{`}, [\![Q + 1]\!]))))
\end{array}
$$

Expanding the $[\![\cdots]\!]$s results in:

$$
\begin{array}{l}
\mathsf{Seq} \\
\quad (\mathsf{Assign}(\text{`}R\text{`}, \lambda s.\ s\text{`}X\text{`}), \\
\quad\quad \mathsf{Seq} \\
\quad\quad\quad (\mathsf{Assign}(\text{`}Q\text{`}, \lambda s.\ 0), \\
\quad\quad\quad\quad \mathsf{While} \\
\quad\quad\quad\quad\quad ((\lambda s.\ s\text{`}Y\text{`} \leq s\text{`}R\text{`}), \\
\quad\quad\quad\quad\quad\quad \mathsf{Seq} \\
\quad\quad\quad\quad\quad\quad\quad (\mathsf{Assign}(\text{`}R\text{`}, \lambda s.\ s\text{`}R\text{`} - s\text{`}Y\text{`}), \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathsf{Assign}(\text{`}Q\text{`}, \lambda s.\ s\text{`}Q\text{`} + 1)))))
\end{array}
$$

□

It might appear that by representing the meaning of commands with relations, we can give a semantics to nondeterministic constructs. For example, if $\mathcal{C}_1 \parallel \mathcal{C}_2$ is the nondeterministic choice 'either do $\mathcal{C}_1$ or do $\mathcal{C}_2$', then one might think that a satisfactory semantics would be given by:

$$[\![\mathcal{C}_1 \parallel \mathcal{C}_2]\!] \;=\; \mathsf{Choose}([\![\mathcal{C}_1]\!], [\![\mathcal{C}_2]\!])$$

where the constant Choose is defined by:

$$\mathsf{Choose}(c_1, c_2)(s_1, s_2) \;=\; c_1(s_1, s_2) \;\vee\; c_2(s_1, s_2)$$

Unfortunately this semantics has some undesirable properties. For example, if c_{while} is the predicate denoted by the non-terminating command **while** T **do skip**, then

$$\forall s_1\ s_2.\ \mathsf{c}_{while}(s_1, s_2) = \mathsf{F}$$

and hence, because $\forall t.\ t \vee \mathsf{F} = t$, it follows that:

$$\textbf{skip} \parallel \mathsf{c}_{while} \;=\; \textbf{skip}$$

Thus the command that does nothing is equivalant to a command that *either* does nothing *or* loops! It is well known how to distinguish guaranteed termination from possible termination [31]; the example above shows that the relational semantics used in this paper does not do it. This problem will appear again in connection with Dijkstra's theory of weakest preconditions in Section 11.2.

5.2 Semantics of partial correctness specifications

A partial correctness specification $\{\mathcal{P}\}\ \mathcal{C}\ \{\mathcal{Q}\}$ denotes:

$$\forall s_1\ s_2.\ [\![\mathcal{P}]\!]\ s_1\ \wedge\ [\![\mathcal{C}]\!](s_1, s_2)\ \Rightarrow\ [\![\mathcal{Q}]\!]\ s_2$$

To abbreviate this formula, we define a constant Spec by:

$$\mathsf{Spec}(p, c, q)\ =\ \forall s_1\ s_2.\ p\ s_1\ \wedge\ c(s_1, s_2)\ \Rightarrow\ q\ s_2$$

Note that the denotation of pre and postconditions $\mathcal{P}$ and $\mathcal{Q}$ are not just the logical formulae themselves, but are $[\![\mathcal{P}]\!]$ and $[\![\mathcal{Q}]\!]$. For example, in the specification $\{X = 1\}\ \mathcal{C}\ \{\mathcal{Q}\}$, the precondition $X = 1$ asserts that the value of the string 'X' in the initial state is 1. The precondition thus denotes $[\![\mathcal{P}]\!]$, i.e. $\lambda s.\ s\text{`}X\text{`} = 1$. Thus:

$$\{X = 1\}\ X := X + 1\ \{X = 2\}$$

denotes

$$\mathsf{Spec}([\![X = 1]\!],\ \mathsf{Assign}(\text{`}X\text{`}, [\![X + 1]\!]),\ [\![X = 2]\!])$$

i.e.

$$\mathsf{Spec}((\lambda s.\ s\text{`}X\text{`} = 1),\ \mathsf{Assign}(\text{`}X\text{`}, \lambda s.\ s\text{`}X\text{`} + 1),\ \lambda s.\ s\text{`}X\text{`} = 2)$$

Example

In the specification below, x and y are logical variables whereas X and Y (and Z) are program variables.

$$\{X = x\ \wedge\ Y = y\}\ Z := X;\ X := Y;\ Y := Z\ \{X = y\ \wedge\ Y = x\}$$

The semantics of this is thus represented by the term:

$$\begin{array}{l}\mathsf{Spec}([\![X = x\ \wedge\ Y = y]\!],\\ \quad \mathsf{Seq}(\mathsf{Assign}(\text{`}Z\text{`}, [\![X]\!]),\\ \quad\quad \mathsf{Seq}(\mathsf{Assign}(\text{`}X\text{`}, [\![Y]\!]), \mathsf{Assign}(\text{`}Y\text{`}, [\![Z]\!]))),\\ \quad [\![X = y\ \wedge\ Y = x]\!])\end{array}$$

which abbreviates:

$$\begin{array}{l}\mathsf{Spec}((\lambda s.\ s\text{`}X\text{`} = x\ \wedge\ s\text{`}Y\text{`} = y),\\ \quad \mathsf{Seq}(\mathsf{Assign}(\text{`}Z\text{`}, \lambda s.\ s\text{`}X\text{`}),\\ \quad\quad \mathsf{Seq}(\mathsf{Assign}(\text{`}X\text{`}, \lambda s.\ s\text{`}Y\text{`}), \mathsf{Assign}(\text{`}Y\text{`}, \lambda s.\ s\text{`}Z\text{`}))),\\ \quad \lambda s.\ s\text{`}X\text{`} = y\ \wedge\ s\text{`}Y\text{`} = x)\end{array}$$

□

6 Hoare logic as higher order logic

Hoare logic can be embedded in higher order logic simply by regarding the concrete syntax given in Sections 2 and 3 as an *abbreviation* for the corresponding semantic formulae described in Section 5. For example:

$$\{X = x\}\ X := X + 1\ \{X = x + 1\}$$

can be interpreted as as abbreviating:

$$\mathsf{Spec}([\![X = x]\!],\ \mathsf{Assign}(`X`, [\![X + 1]\!]),\ [\![X = x + 1]\!])$$

i.e.

$$\mathsf{Spec}((\lambda s.\ s`X` = x),\ \mathsf{Assign}(`X`, \lambda s.\ s`X` + 1),\ \lambda s.\ s`X` = x + 1)$$

The translation between the syntactic 'surface structure' and the semantic 'deep structure' is straightforward; it can easily be mechanized with a simple parser and pretty-printer. Section 7 contains examples illustrating this in a version of the HOL system.

If partial correctness specifications are interpreted this way then, as shown in the rest of this section, the axioms and rules of Hoare logic described in Section 3 become derived rules of higher order logic.

The first step in this derivation is to prove the following seven theorems from the definitions of the constants Spec, Skip, Assign, Bnd, Seq, If, While and Iter.

H1. $\vdash \forall p.\ \mathsf{Spec}(p, \mathsf{Skip}, p)$

H2. $\vdash \forall p\ v\ e.\ \mathsf{Spec}((\lambda s.\ p(\mathsf{Bnd}(e\ s, v, s))), \mathsf{Assign}(v, e), p)$

H3. $\vdash \forall p\ p'\ q\ c.\ (\forall s.\ p'\ s\ \Rightarrow\ p\ s)\ \wedge\ \mathsf{Spec}(p, c, q)\ \Rightarrow\ \mathsf{Spec}(p', c, q)$

H4. $\vdash \forall p\ q\ q'\ c.\ \mathsf{Spec}(p, c, q)\ \wedge\ (\forall s.\ q\ s\ \Rightarrow\ q'\ s)\ \Rightarrow\ \mathsf{Spec}(p, c, q')$

H5. $\vdash \forall p\ q\ r\ c_1\ c_2.\ \mathsf{Spec}(p, c_1, q)\ \wedge\ \mathsf{Spec}(q, c_2, r)\ \Rightarrow\ \mathsf{Spec}(p, \mathsf{Seq}(c_1, c_2), r)$

H6. $\vdash \forall p\ q\ c_1\ c_2\ b.$
$\mathsf{Spec}((\lambda s.\ p\ s\ \wedge\ b\ s), c_1, q)\ \wedge\ \mathsf{Spec}((\lambda s.\ p\ s\ \wedge\ \neg(b\ s)), c_2, q)$
$\Rightarrow$
$\mathsf{Spec}(p, \mathsf{If}(b, c_1, c_2), q)$

H7. $\vdash \forall p\ c\ b.$
$\mathsf{Spec}((\lambda s.\ p\ s\ \wedge\ b\ s), c, p)$
$\Rightarrow$
$\mathsf{Spec}(p, \mathsf{While}(b, c), (\lambda s.\ p\ s\ \wedge\ \neg(b\ s)))$

The proofs of H1 to H7 are routine. All the axioms and rules of Hoare logic, *except* for the assignment axiom, can be implemented in a uniform way from H1 – H7. The derivation of the assignment axiom from H2, although straightforward, is a bit messy; it is thus explained last (in Section 6.7).

6.1 Derivation of the skip-axiom

To derive the **skip**-axiom it must be shown for arbitrary $\mathcal{P}$ that:

$$\vdash\ \{\mathcal{P}\}\ \mathbf{skip}\ \{\mathcal{P}\}$$

which abbreviates:

$$\vdash\ \mathsf{Spec}([\![\mathcal{P}]\!], \mathsf{Skip}, [\![\mathcal{P}]\!])$$

This follows by specializing p to $[\![\mathcal{P}]\!]$ in H1.

6.2 Derivation of precondition strengthening

To derive the rule of precondition strengthening it must be shown that for arbitrary $\mathcal{P}$, $\mathcal{P}'$, $\mathcal{C}$ and $\mathcal{Q}$ that:

$$\frac{\vdash\ \mathcal{P}'\ \Rightarrow\ \mathcal{P} \qquad \vdash\ \{\mathcal{P}\}\ \mathcal{C}\ \{\mathcal{Q}\}}{\vdash\ \{\mathcal{P}'\}\ \mathcal{C}\ \{\mathcal{Q}\}}$$

Expanding abbreviations converts this to:

$$\frac{\vdash\ \mathcal{P}'\ \Rightarrow\ \mathcal{P} \qquad \vdash\ \mathsf{Spec}([\![\mathcal{P}]\!], [\![\mathcal{C}]\!], [\![\mathcal{Q}]\!])}{\vdash\ \mathsf{Spec}([\![\mathcal{P}']\!], [\![\mathcal{C}]\!], [\![\mathcal{Q}]\!])}$$

Specializing H3 yields:

$$\vdash\ (\forall s.\ [\![\mathcal{P}']\!]\ s\ \Rightarrow\ [\![\mathcal{P}]\!]\ s)\ \wedge\ \mathsf{Spec}([\![\mathcal{P}]\!], [\![\mathcal{C}]\!], [\![\mathcal{Q}]\!])\ \Rightarrow\ \mathsf{Spec}([\![\mathcal{P}']\!], [\![\mathcal{C}]\!], [\![\mathcal{Q}]\!])$$

The rule of precondition strengthening will follow if $\vdash\ \forall s.\ [\![\mathcal{P}']\!]\ s\ \Rightarrow\ [\![\mathcal{P}]\!]\ s$ can be deduced from $\vdash\ \mathcal{P}'\ \Rightarrow\ \mathcal{P}$. To see that this is indeed the case, let us make explicit the program variables $X_1, \ldots, X_n$ occurring in $\mathcal{P}$ and $\mathcal{P}'$ by writing $\mathcal{P}[X_1, \ldots, X_n]$ and $\mathcal{P}[X_1, \ldots, X_n]'$. Then $\vdash\ \mathcal{P}'\ \Rightarrow\ \mathcal{P}$ becomes

$$\vdash\ \mathcal{P}'[X_1, \ldots, X_n]\ \Rightarrow\ \mathcal{P}[X_1, \ldots, X_n]$$

Since $X_1, \ldots, X_n$ are free variables in this theorem, they are implicitly universally quantified, and hence each X_i can be instantiated to $s`X_i`$ to get:

$$\vdash\ \mathcal{P}'[s`X_1`, \ldots, s`X_n`]\ \Rightarrow\ \mathcal{P}[s`X_1`, \ldots, s`X_n`]$$

Generalizing on the free variable s yields:

$$\vdash\ \forall s.\ \mathcal{P}'[s`X_1`, \ldots, s`X_n`]\ \Rightarrow\ \mathcal{P}[s`X_1`, \ldots, s`X_n`]$$

which is equivalent (by β-reduction) to

$$\vdash\ \forall s.\ (\lambda s.\ \mathcal{P}'[s`X_1`, \ldots, s`X_n`])\ s\ \Rightarrow\ (\lambda s.\ \mathcal{P}[s`X_1`, \ldots, s`X_n`])\ s$$

i.e.

$$\vdash\ \forall s.\ [\![\mathcal{P}'[X_1, \ldots, X_n]]\!]\ s\ \Rightarrow\ [\![\mathcal{P}[X_1, \ldots, X_n]]\!]\ s$$

The derivation sketched above can be done via higher order matching applied to H3. This is, in fact, how the HOL derived rule `PRE_STRENGTH_RULE` described in Section 7 is programmed. Systems like *Isabelle* [29], which have higher order unification built in, would handle such rules very smoothly.

6.3 Derivation of postcondition weakening

To derive the rule of postcondition weakening, it must be shown that for arbitrary $\mathcal{P}$, $\mathcal{C}$, and $\mathcal{Q}$ and $\mathcal{Q}'$ that:

$$\frac{\vdash\ \{\mathcal{P}\}\ \mathcal{C}\ \{\mathcal{Q}\} \qquad \vdash\ \mathcal{Q}\ \Rightarrow\ \mathcal{Q}'}{\vdash\ \{\mathcal{P}\}\ \mathcal{C}\ \{\mathcal{Q}'\}}$$

The derivation of this from H4 is similar to the derivation of precondition strengthening from H3.

6.4 Derivation of the sequencing rule

To derive the sequencing rule, it must be shown that for arbitrary $\mathcal{P}$, $\mathcal{C}_1$, $\mathcal{R}$, $\mathcal{C}_2$ and and $\mathcal{Q}$ that:

$$\frac{\vdash\ \{\mathcal{P}\}\ \mathcal{C}_1\ \{\mathcal{Q}\} \qquad \vdash\ \{\mathcal{Q}\}\ \mathcal{C}_2\ \{\mathcal{R}\}}{\vdash\ \{\mathcal{P}\}\ \mathcal{C}_1 ;\ \mathcal{C}_2\ \{\mathcal{R}\}}$$

Expanding the abbreviations yields:

$$\frac{\vdash\ \mathsf{Spec}([\![\mathcal{P}]\!], [\![\mathcal{C}_1]\!], [\![\mathcal{Q}]\!]) \qquad \vdash\ \mathsf{Spec}([\![\mathcal{Q}]\!], [\![\mathcal{C}_2]\!], [\![\mathcal{R}]\!])}{\vdash\ \mathsf{Spec}([\![\mathcal{P}]\!], \mathsf{Seq}([\![\mathcal{C}_1]\!], [\![\mathcal{C}_2]\!]), [\![\mathcal{R}]\!])}$$

The validity of this rule follows directly from H5.

6.5 Derivation of the if-rule

To derive the **if**-rule, it must be shown that for arbitrary $\mathcal{P}$, $\mathcal{B}$, $\mathcal{C}_1$, $\mathcal{C}_2$ and and $\mathcal{Q}$ that:

$$\frac{\vdash\ \{\mathcal{P}\ \wedge\ \mathcal{B}\}\ \mathcal{C}_1\ \{\mathcal{Q}\} \qquad \vdash\ \{\mathcal{P}\ \wedge\ \neg\mathcal{B}\}\ \mathcal{C}_2\ \{\mathcal{Q}\}}{\vdash\ \{\mathcal{P}\}\ \textbf{if}\ \mathcal{B}\ \textbf{then}\ \mathcal{C}_1\ \textbf{else}\ \mathcal{C}_2\ \{\mathcal{Q}\}}$$

Expanding abbreviations yields:

$$\frac{\vdash\ \mathsf{Spec}([\![\mathcal{P}\ \wedge\ \mathcal{B}]\!], [\![\mathcal{C}_1]\!], [\![\mathcal{Q}]\!]) \qquad \vdash\ \mathsf{Spec}([\![\mathcal{P}\ \wedge\ \neg\mathcal{B}]\!], [\![\mathcal{C}_2]\!], [\![\mathcal{Q}]\!])}{\vdash\ \mathsf{Spec}([\![\mathcal{P}]\!], \mathsf{If}([\![\mathcal{B}]\!], [\![\mathcal{C}_1]\!], [\![\mathcal{C}_2]\!]), [\![\mathcal{Q}]\!])}$$

This follows from H6 in a similar fashion to the way precondition strenthening follows from H3.

6.6 Derivation of the while-rule

To derive the **while**-rule, it must be shown that for arbitrary $\mathcal{P}$, $\mathcal{B}$ and $\mathcal{C}$ that:

$$\frac{\vdash\ \{\mathcal{P}\ \wedge\ \mathcal{B}\}\ \mathcal{C}\ \{\mathcal{P}\}}{\vdash\ \{\mathcal{P}\}\ \textbf{while}\ \mathcal{B}\ \textbf{do}\ \mathcal{C}\ \{\mathcal{P}\ \wedge\ \neg\mathcal{B}\}}$$

Expanding abbreviations yields:

$$\frac{\vdash\ \mathsf{Spec}([\![\mathcal{P}\ \wedge\ \mathcal{B}]\!], [\![\mathcal{C}]\!], [\![\mathcal{P}]\!])}{\vdash\ \mathsf{Spec}([\![\mathcal{P}]\!], \mathsf{While}([\![\mathcal{B}]\!], [\![\mathcal{C}]\!]), [\![\mathcal{P}\ \wedge\ \neg\mathcal{B}]\!])}$$

This follows from H7.

6.7 Derivation of the assignment axiom

To derive the assignment axiom, it must be shown that for arbitrary $\mathcal{P}$, $\mathcal{E}$ and $\mathcal{V}$:

$$\vdash \ \{\mathcal{P}\,[\mathcal{E}/\mathcal{V}]\}\ \mathcal{V} := \mathcal{E}\ \{\mathcal{P}\}$$

This abbreviates:

$$\vdash \ \mathsf{Spec}([\![\mathcal{P}\,[\mathcal{E}/\mathcal{V}]\,]\!], \mathsf{Assign}(‘\mathcal{V}‘, [\![\mathcal{E}]\!]), [\![\mathcal{P}]\!])$$

By H2:

$$\vdash \ \forall p\ x\ e.\ \mathsf{Spec}((\lambda s.\ p(\mathsf{Bnd}(e\ s, x, s))), \mathsf{Assign}(x, e), p)$$

Specializing p, x and e to $[\![P]\!]$, ‘$\mathcal{V}$‘ and $[\![\mathcal{E}]\!]$ yields:

$$\vdash \ \mathsf{Spec}((\lambda s.\ [\![P]\!](\mathsf{Bnd}([\![\mathcal{E}]\!]s, ‘\mathcal{V}‘, s))), \mathsf{Assign}(‘\mathcal{V}‘, [\![\mathcal{E}]\!]), [\![P]\!])$$

Thus, to derive the assignment axiom it must be shown that:

$$\vdash \ [\![\mathcal{P}\,[\mathcal{E}/\mathcal{V}]\,]\!] \ = \ \lambda s.\ [\![P]\!](\mathsf{Bnd}([\![\mathcal{E}]\!]s, ‘\mathcal{V}‘, s))$$

To see why this holds, let us make explicit the free program variables in $\mathcal{P}$ and $\mathcal{E}$ by writing $\mathcal{P}[\mathcal{V}, X_1, \ldots, X_n]$ and $\mathcal{E}[\mathcal{V}, X_1, \ldots, X_n]$, where X_1, ... , X_n are the free program variables that are not equal to $\mathcal{V}$. Then, for example, $\mathcal{P}[1, \ldots, n]$ would denote the result of substituting 1, ... , n for X_1, ... , X_n in $\mathcal{P}$ respectively. The equation above thus becomes:

$$\begin{array}{l} [\![\mathcal{P}[\mathcal{V}, X_1, \ldots, X_n]\,[\mathcal{E}[\mathcal{V}, X_1, \ldots, X_n]/\mathcal{V}]\,]\!] \\ = \\ \lambda s.\ [\![\mathcal{P}[\mathcal{V}, X_1, \ldots, X_n]]\!](\mathsf{Bnd}([\![\mathcal{E}[\mathcal{V}, X_1, \ldots, X_n]]\!]s, ‘\mathcal{V}‘, s)) \end{array}$$

Performing the substitution in the left hand side yields:

$$\begin{array}{l} [\![\mathcal{P}[\mathcal{E}[\mathcal{V}, X_1, \ldots, X_n], X_1, \ldots, X_n]]\!] \\ = \\ \lambda s.\ [\![\mathcal{P}[\mathcal{V}, X_1, \ldots, X_n]]\!](\mathsf{Bnd}([\![\mathcal{E}[\mathcal{V}, X_1, \ldots, X_n]]\!]s, ‘\mathcal{V}‘, s)) \end{array}$$

Replacing expressions of the form $[\![\mathcal{P}[\cdots]]\!]$ by their meaning yields:

$$\begin{array}{l} (\lambda s.\ \mathcal{P}[\mathcal{E}[s‘\mathcal{V}‘, s‘X_1‘, \ldots, s‘X_n‘], s‘X_1‘, \ldots, s‘X_n‘]) \\ = \\ \lambda s.\ (\lambda s.\ \mathcal{P}[s‘\mathcal{V}‘, s‘X_1‘, \ldots, s‘X_n‘])(\mathsf{Bnd}([\![\mathcal{E}[\mathcal{V}, X_1, \ldots, X_n]]\!]s, ‘\mathcal{V}‘, s)) \end{array}$$

Performing a β-reduction on the right hand side, and then simplifying with the following easily derived properties of Bnd (the second of which assumes ‘$\mathcal{V}$‘ $\neq X_i$):

$$\vdash \ \mathsf{Bnd}([\![\mathcal{E}[\mathcal{V}, X_1, \ldots, X_n]]\!]s, ‘\mathcal{V}‘, s)\ ‘\mathcal{V}‘ \ = \ [\![\mathcal{E}[\mathcal{V}, X_1, \ldots, X_n]]\!]s$$

$$\vdash \ \mathsf{Bnd}([\![\mathcal{E}[\mathcal{V}, X_1, \ldots, X_n]]\!]s, ‘\mathcal{V}‘, s)\ X_i \ = \ s\ X_i$$

results in:

$$\begin{array}{l} (\lambda s.\ \mathcal{P}[\mathcal{E}[s‘\mathcal{V}‘, s‘X_1‘, \ldots, s‘X_n‘], s‘X_1‘, \ldots, s‘X_n‘]) \\ = \\ \lambda s.\ \mathcal{P}[[\![\mathcal{E}[\mathcal{V}, X_1, \ldots, X_n]]\!]s, s‘X_1‘, \ldots, s‘X_n‘] \end{array}$$

which is true since:

$$[\![\mathcal{E}[\mathcal{V}, X_1, \ldots, X_n]]\!]s \ = \ \mathcal{E}[s‘\mathcal{V}‘, s‘X_1‘, \ldots, s‘X_n‘]$$

Although this derivation might appear tricky at first sight, it is straightforward and easily mechanized. The HOL derived rule `ASSIGN_AX` described in Section 7 performs this deduction for each $\mathcal{P}$, $\mathcal{E}$ and $\mathcal{V}$.

It is tempting to try to formulate the assignment axiom as a theorem of higher order logic looking something like:

$$\forall p\ e\ v.\ \mathsf{Spec}(p\,\texttt{[}e/v\texttt{]}\,, \mathsf{Assign}(v, e), p)$$

Unfortunately, the expression $p\,\texttt{[}e/v\texttt{]}$ does not make sense when p is a variable. $\mathcal{P}\,\texttt{[}\mathcal{E}/\mathcal{V}\texttt{]}$ is a meta notation and consequently the assignment axiom can only be stated as a meta theorem. This elementary point is nevertheless quite subtle. In order to prove the assignment axiom as a theorem within higher order logic it would be necessary to have types in the logic corresponding to formulae, variables and terms. One could then prove something like:

$$\forall P\ E\ V.\ \mathsf{Spec}(\mathsf{Truth}(\mathsf{Subst}(P, E, V)),\ \mathsf{Assign}(V, \mathsf{Value}\ E),\ \mathsf{Truth}\ P)$$

It is clear that working out the details of this would be lots of work. This sort of embedding of a subset of a logic within itself has been explored in the context of the Boyer-Moore theorem prover [3].

7 Hoare logic in HOL

In this section, the mechanization of the axioms and rules of Hoare logic using the HOL system [13] will be illustrated. We will try to make this comprehensible to readers unfamiliar with HOL, but it would help to have some prior exposure to Milner's LCF approach to interactive proof [10, 11].

The HOL system is a version of Cambridge LCF [30] with higher order logic as object language[12]. Cambridge LCF evolved from Edinburgh LCF [11]. Some of the material in this and following sections has been taken from the paper *A Proof Generating System for Higher Order Logic* [13].

The boxes below contain a session with a version of the HOL system that has been extended to support Hoare logic using the approach described in Section 6. The actual code implementing the extensions is not described here; it is mostly just straightforward ML (but the parser and pretty-printer are implemented in Lisp). To help the reader, the transcripts of the sessions with HOL have been edited so that proper logical symbols appear instead of their ASCII representations[13].

The user interface to HOL (and LCF) is the interactive programming language ML. At top level, expressions can be evaluated and declarations performed. The former results in the value of the expression and it ML type being printed; the latter in a value being bound to a name. The interactions in the boxes that follow should be understood as occurring in sequence. For example, variable bindings made in earlier boxes are assumed to persist to later ones. The ML prompt is `#`, so lines beginning with `#` are typed by the user and other lines are the system's response.

The ML language has a similar type system to the one used by the HOL logic. It is very common to confuse the two type systems. In this paper `small teletype` font

[12]LCF has the Scott/Milner 'Logic of Computable Functions' as object language.

[13]An interface to HOL that supports proper logical characters has been implemented by Roger Jones of ICL Defence Systems. This interface is also used to support a surface syntax based on the Z notation [33]. ICL use HOL for the specification and verification of security properties of both hardware and software.

will be used for ML types and *slanted* font for logical types. For example, the ML expression `1` has ML type `int`, whereas the HOL constant 1 has logical type *num*.

The first box below illustrates how the parser and pretty-printer have been modified to translate between 'surface' and 'deep' structure. The curly brackets { and } function like ⟦ and ⟧ of Section 5; i.e. "$\{\mathcal{F}[\mathtt{X}_1,\ldots,\mathtt{X}_n]\}$" parses to "$\lambda\mathtt{s}.\mathcal{F}[\mathtt{s}`\mathtt{X}_1`,\ldots,\mathtt{s}`\mathtt{X}_n`]$". Evaluating `pretty_on()` switches the pretty-printer on; it can be switched off by evaluating `pretty_off()`.

```
#"{(R=x) ∧ (Y=y)}" ;;
"λs. (s ‘R‘ = x) ∧ (s ‘Y‘ = y)" : term

#"R:=X" ;;
"Assign(‘R‘,(λs. s ‘X‘))" : term

#pretty_on();;
() : void

#"R:=X" ;;
"R := X" : term
```

The ML function `ASSIGN_AX` has ML type `term -> term -> thm` and implements the assignment axiom.

$$\texttt{ASSIGN_AX "}\{\mathcal{P}\}\texttt{" "}\mathcal{V}\texttt{ := }\mathcal{E}\texttt{"} \quad\mapsto\quad \vdash\ \{\mathcal{P}[\mathcal{E}/\mathcal{V}]\}\ \mathcal{V}\ \texttt{:=}\ \mathcal{E}\ \{\mathcal{P}\}$$

```
#let hth1 = ASSIGN_AX "{(R=x) ∧ (Y=y)}" "R:=X" ;;
hth1 = ⊢ {(X = x) ∧ (Y = y)} R := X {(R = x) ∧ (Y = y)}

#let hth2 = ASSIGN_AX "{(R=x) ∧ (X=y)}" "X:=Y" ;;
hth2 = ⊢ {(R = x) ∧ (Y = y)} X := Y {(R = x) ∧ (X = y)}

#let hth3 = ASSIGN_AX "{(Y=x) ∧ (X=y)}" "Y:=R" ;;
hth3 = ⊢ {(R = x) ∧ (X = y)} Y := R {(Y = x) ∧ (X = y)}
```

The ML function `SEQ_RULE` has ML type `thm # thm -> thm` and implements the sequencing rule.

$$\texttt{SEQ_RULE}\ (\vdash\ \{\mathcal{P}\}\ \mathcal{C}_1\ \{\mathcal{Q}\},\ \ \vdash\ \{\mathcal{Q}\}\ \mathcal{C}_2\ \{\mathcal{R}\}) \quad\mapsto\quad \vdash\ \{\mathcal{P}\}\ \mathcal{C}_1;\ \mathcal{C}_2\ \{\mathcal{R}\}$$

```
#let hth4 = SEQ_RULE (hth1,hth2);;
hth4 = ⊢ {(X = x) ∧ (Y = y)} R := X; X := Y {(R = x) ∧ (X = y)}

#let hth5 = SEQ_RULE (hth4,hth3);;
hth5 =
⊢ {(X = x) ∧ (Y = y)}
    R := X; X := Y; Y := R
  {(Y = x) ∧ (X = y)}
```

If the pretty printing is switched off, the actual terms being manipulated become visible.

```
#pretty_off();;
() : void

#hth5;;
⊢ Spec
   ((λs. (s `X` = x) ∧ (s `Y` = y)),
    Seq
    (Seq(Assign(`R`,(λs. s `X`)),Assign(`X`,(λs. s `Y`))),
     Assign(`Y`,(λs. s `R`))),(λs. (s `Y` = x) ∧ (s `X` = y)))

#pretty_on();;
() : void
```

Using ML it is easy to define a function SEQL_RULE of type thm list -> thm that implements a derived rule generalizing SEQ_RULE from two arguments to a list of arguments.

$$\begin{array}{l}\texttt{SEQL_RULE} \\ \;\texttt{[}\vdash\ \{\mathcal{P}\}\ \mathcal{C}_1\ \{\mathcal{Q}_1\}\texttt{;}\ \ \vdash\ \{\mathcal{Q}_1\}\ \mathcal{C}_2\ \{\mathcal{Q}_2\}\texttt{;}\ \ldots\ \texttt{;}\ \ \vdash\ \{\mathcal{Q}_{n-1}\}\ \mathcal{C}_n\ \{\mathcal{R}\}\texttt{]} \\ \;\mapsto \\ \;\vdash\ \{\mathcal{P}\}\ \mathcal{C}_1\texttt{;}\ldots\texttt{;}\mathcal{C}_n\ \{\mathcal{R}\}\end{array}$$

For readers familiar with ML, here is the definition of SEQL_RULE.

```
#letrec SEQL_RULE thl =
# if null(tl thl) then hd thl
#                  else SEQL_RULE
#                         (SEQ_RULE(hd thl,hd(tl thl)).tl(tl thl))
SEQL_RULE = - : proof

#let hth6 = SEQL_RULE[hth1;hth2;hth3];;
hth6 =
⊢ {(X = x) ∧ (Y = y)}
    R := X; X := Y; Y := R
  {(Y = x) ∧ (X = y)}
```

The ML function PRE_STRENGTH_RULE has type thm # thm -> thm and implements the rule of precondition strengthening.

$$\texttt{PRE_STRENGTH_RULE}\ (\vdash\ \mathcal{P}'\ \Rightarrow\ \mathcal{P},\ \vdash\ \{\mathcal{P}\}\ \mathcal{C}\ \{\mathcal{Q}\})\ \mapsto\ \vdash\ \{\mathcal{P}'\}\ \mathcal{C}\ \{\mathcal{Q}\}$$

The ML function POST_WEAK_RULE has type thm # thm -> thm implements the rule of postcondition weakening.

$$\texttt{POST_WEAK_RULE}\ (\vdash\ \{\mathcal{P}\}\ \mathcal{C}\ \{\mathcal{Q}\},\ \vdash\ \mathcal{Q}\ \Rightarrow\ \mathcal{Q}')\ \mapsto\ \vdash\ \{\mathcal{P}'\}\ \mathcal{C}\ \{\mathcal{Q}\}$$

POST_WEAK_RULE is illustrated in the sessions with HOL below.

In the box below, the predefined constant MAX and lemma MAX_LEMMA1 are used.

```
#MAX;;
⊢ MAX(m,n) = (m > n → m | n)

#let hth7 = ASSIGN_AX "{R = MAX(x,y)}" "R := Y" ;;
hth7 = ⊢ {Y = MAX(x,y)} R := Y {R = MAX(x,y)}

#MAX_LEMMA1;;
⊢ ((X = x) ∧ (Y = y)) ∧ Y > X ⇒ (Y = MAX(x,y))

#let hth8 = PRE_STRENGTH_RULE(MAX_LEMMA1,hth7);;
hth8 = ⊢ {((X = x) ∧ (Y = y)) ∧ Y > X} R := Y {R = MAX(x,y)}
```

The ML function `IF_RULE` has type `thm # thm -> thm` and implements the **if**-rule.

$$\texttt{IF_RULE}\ (\vdash\ \{\mathcal{P}\ \wedge\ \mathcal{B}\}\ \mathcal{C}_1\ \{\mathcal{Q}\},\ \vdash\ \{\mathcal{P}\ \wedge\ \neg\mathcal{B}\}\ \mathcal{C}_2\ \{\mathcal{Q}\})$$
$$\mapsto$$
$$\vdash\ \{\mathcal{P}\}\ \texttt{if}\ \mathcal{B}\ \texttt{then}\ \mathcal{C}_1\ \texttt{else}\ \mathcal{C}_2\ \ \{\mathcal{Q}\}$$

`MAX_LEMMA2` is used in the next box; it is a pre-proved lemma about `MAX`.

```
#let hth9 = ASSIGN_AX "{R = MAX(x,y)}" "R := X" ;;
hth9 = ⊢ {X = MAX(x,y)} R := X {R = MAX(x,y)}

#MAX_LEMMA2;;
⊢ ((X = x) ∧ (Y = y)) ∧ ¬ Y > X ⇒ (X = MAX(x,y))

#let hth10 = PRE_STRENGTH_RULE(MAX_LEMMA2,hth9);;
hth10 = ⊢ {((X = x) ∧ (Y = y)) ∧ ¬ Y > X} R := X {R = MAX(x,y)}

#let hth11 = IF_RULE(hth8,hth10);;
hth11 =
⊢ {(X = x) ∧ (Y = y)}
    if Y > X then R := Y else R := X
  {R = MAX(x,y)}
```

The ML function `WHILE_RULE` has type `thm -> thm` and implements the **while**-rule.

$$\texttt{WHILE_RULE}\ \vdash\ \{\mathcal{P}\ \wedge\ \mathcal{B}\}\ \mathcal{C}\ \{\mathcal{P}\}\ \mapsto\ \vdash\ \{\mathcal{P}\}\ \texttt{while}\ \mathcal{B}\ \texttt{do}\ \mathcal{C}\ \ \{\mathcal{P}\ \wedge\ \neg\mathcal{B}\}$$

To illustrate the **while**-rule, a HOL transcript of the example proof in Section 3.2 is now given. The **while**-rule is used to prove `Th9` below. For completeness, all the proofs of the pure logic theorems that are needed are included. These are performed using tactics; readers unfamiliar with tactics should either skip the proofs of `Th7`, `Th11` and `Th12`, or read Section 8 below.

```
#let Th1 = ASSIGN_AX "{X = R + (Y × 0)}" "R := X" ;;
Th1 = ⊢ {X = X + (Y × 0)} R := X {X = R + (Y × 0)}

#let Th2 = ASSIGN_AX "{X = R + (Y × Q)}" "Q := 0" ;;
Th2 = ⊢ {X = R + (Y × 0)} Q := 0 {X = R + (Y × Q)}

#let Th3 = SEQ_RULE(Th1,Th2);;
Th3 = ⊢ {X = X + (Y × 0)} R := X; Q := 0 {X = R + (Y × Q)}
```

```
#let Th4 = ASSIGN_AX "{X = R + (Y × (Q + 1))}" "R := (R − Y)" ;;
Th4 =
⊢ {X = (R − Y) + (Y × (Q + 1))}
    R := R − Y
  {X = R + (Y × (Q + 1))}

#let Th5 = ASSIGN_AX "{X = R + (Y × Q)}" "Q := (Q + 1)" ;;
Th5 = ⊢ {X = R + (Y × (Q + 1))} Q := Q + 1 {X = R + (Y × Q)}

#let Th6 = SEQ_RULE(Th4,Th5);;
Th6 =
⊢ {X = (R − Y) + (Y × (Q + 1))}
    R := R − Y; Q := Q + 1
  {X = R + (Y × Q)}
```

In the next box, a simple arithmetical fact called `Th7` is proved. Some systems can prove such facts fully automatically [2]; unfortunately this is not so with HOL and the user must supply a proof outline expressed as a *tactic*. Tactics are described in Section 8 below, and readers unfamiliar with them might want to read that section now. To make this paper less dependent on detailed knowledge of HOL's particular repertoire of tactics, some of the ML code used in the sessions that follow has been replaced by English descriptions. For example, the actual code for proving `Th7` is:

```
#let Th7 =
# TAC_PROOF
#  (([], "(X = R + (Y * Q)) /\ (Y <= R)
#          ==> (X = (R - Y) + (Y * (Q + 1)))"),
#   REPEAT STRIP_TAC
#    THEN REWRITE_TAC[LEFT_ADD_DISTRIB;MULT_CLAUSES]
#    THEN ONCE_REWRITE_TAC[SPEC "Y*Q" ADD_SYM]
#    THEN ONCE_REWRITE_TAC[ADD_ASSOC]
#    THEN IMP_RES_TAC SUB_ADD
#    THEN ASM_REWRITE_TAC[]);;

Th7 = |- (X = R + (Y * Q)) /\ Y <= R ==> (X = (R - Y) + (Y * (Q + 1)))
```

which can be read informally as:

```
#let Th7 =
# TAC_PROOF
#  (([], "(X = R + (Y × Q)) ∧ (Y ≤ R)
#          ⇒ (X = (R − Y) + (Y × (Q + 1)))"),
#    'Move conjuncts of antecedent of implication to assumption list'
#    THEN 'Simplify using lemmas about + and ×'
#    THEN 'Expand assumptions using ⊢ n ≤ m ⇒ (m − n) + n = m'
#    THEN 'Simplify using assumptions' );;

Th7 = ⊢ (X = R + (Y × Q)) ∧ Y ≤ R ⇒ (X = (R − Y) + (Y × (Q + 1)))
```

Subsequent sessions will contain similar informal English descriptions of tactics, rather than exact ML code.

Continuing our session: if `Th7` is used to strengthen the precondition of `Th6`, the result is then a suitable hypothesis for the **while**-rule.

```
#let Th8 = PRE_STRENGTH_RULE(Th7,Th6);;
Th8 =
⊢ {(X = R + (Y × Q)) ∧ Y ≤ R}
    R := R − Y; Q := Q + 1
  {X = R + (Y × Q)}

#let Th9 = WHILE_RULE Th8;;
Th9 =
⊢ {X = R + (Y × Q)}
    while Y ≤ R do R := R − Y; Q := Q + 1
  {(X = R + (Y × Q)) ∧ ¬ Y ≤ R}

#let Th10 = SEQ_RULE(Th3,Th9);;
Th10 =
⊢ {X = X + (Y × 0)}
    R := X; Q := 0; while Y ≤ R do R := R − Y; Q := Q + 1
  {(X = R + (Y × Q)) ∧ ¬ Y ≤ R}
```

The proof of `Th10` could have been done in a single complicated step.

```
#SEQL_RULE
# [ASSIGN_AX "{X = R + (Y × 0)}" "R := X";
#  ASSIGN_AX "{X = R + (Y × Q)}" "Q := 0";
#  WHILE_RULE
#   (PRE_STRENGTH_RULE
#     (DISTRIB_LEMMA,
#       SEQL_RULE
#        [ASSIGN_AX "{X = R + (Y × (Q + 1))}" "R := (R − Y)";
#         ASSIGN_AX "{X = R + (Y × Q)}" "Q := (Q + 1)"]))];;
⊢ {X = X + (Y × 0)}
    R := X; Q := 0; while Y ≤ R do R := R − Y; Q := Q + 1
  {(X = R + (Y × Q)) ∧ ¬ Y ≤ R}
```

Two lemmas are now proved that will be used to simplify the precondition and postcondition of `Th10`.

```
#let Th11 =
# TAC_PROOF
#  (([],"T ⇒ (X = X + (Y × 0))"),
#    ‘Simplify’ );;
Th11 = ⊢ T ⇒ (X = X + (Y × 0))

#let Th12 =
# TAC_PROOF
#  (([],"(X = R + (Y × Q)) ∧ ¬ Y ≤ R
#          ⇒ (X = R + (Y × Q)) ∧ R < Y"),
#    ‘Move antecedent of implication to assumption list’
#    THEN ‘Simplify using assumptions and ⊢ ¬(m < n = n ≤ m)’
Th12 = ⊢ (X = R + (Y × Q)) ∧ ¬ Y ≤ R ⇒ (X = R + (Y × Q)) ∧ R < Y
```

The pre and postconditions of `Th10` can now be simplified in a single step.

```
#let Th13 = POST_WEAK_RULE(PRE_STRENGTH_RULE(Th11,Th10),Th12);;
Th13 =
⊢ {T}
    R := X; Q := 0; while Y ≤ R do R := R − Y; Q := Q + 1
  {(X = R + (Y × Q)) ∧ R < Y}
```

This completes the mechanical generation of the proof in Section 3.2.

The proof of `Th13` could be done by rewriting `Th10` without recourse to Hoare logic at all. First the lemma $\vdash \neg(Y \leq R) = (R < Y)$ is proved:

```
#let Th14 =
# TAC_PROOF
#  (([],"¬ (Y ≤ R) = (R < Y)"),
#    ‘Simplify using ⊢ ¬(m < n = n ≤ m’ );;
Th14 = ⊢ ¬ Y ≤ R = R < Y
```

Then `Th10` is rewritten using `Th14` and elementary properties of addition and multiplication.

```
# ‘Simplify Th10 using Th14 and properties of + and ×’ ;;
⊢ {T}
    R := X; Q := 0; while Y ≤ R do R := R − Y; Q := Q + 1
  {(X = R + (Y × Q)) ∧ R < Y}
```

To see how this works let us look at the ‘deep structure’ of `Th10`.

```
#pretty_off();;
() : void

#Th10;;
⊢ Spec
   ((λs. s ‘X‘ = (s ‘X‘) + ((s ‘Y‘) × 0)),
    Seq(Seq(Assign(‘R‘,(λs. s ‘X‘)),Assign(‘Q‘,(λs. 0))),
        While
        ((λs. (s ‘Y‘) ≤ (s ‘R‘)),
         Seq
         (Assign(‘R‘,(λs. (s ‘R‘) − (s ‘Y‘))),
          Assign(‘Q‘,(λs. (s ‘Q‘) + 1))))),
    (λs. (s ‘X‘ = (s ‘R‘) + ((s ‘Y‘) × (s ‘Q‘))) ∧
         ¬ (s ‘Y‘) ≤ (s ‘R‘)))
```

Rewriting `Th10` with $\vdash m \times 0 = 0$ will replace `(s ‘Y‘) × 0` by `0`. Rewriting with $\vdash m + 0 = m$ and `Th14` works similarly.

```
#  'Simplify using Th14 and properties of + and ×' ;;
⊢ Spec
   ((λs. T),
    Seq(Seq(Assign(`R`,(λs. s `X`)),Assign(`Q`,(λs. 0))),
        While
        ((λs. (s `Y`) ≤ (s `R`)),
         Seq
         (Assign(`R`,(λs. (s `R`) − (s `Y`))),
          Assign(`Q`,(λs. (s `Q`) + 1))))),
    (λs. (s `X` = (s `R`) + ((s `Y`) × (s `Q`))) ∧
         (s `R`) < (s `Y`)))
```

Thus although the pretty-printer makes it look as though `Y ×0` is rewritten to `0`, what actually happens is that `(s `Y`) × 0` is rewritten to `0`. This direct application of a HOL theorem proving tool to the semantics of a partial correctness specification illustrates how reasoning with HOL tools can be mixed with reasoning in Hoare logic. We suspect such mixtures of 'axiomatic' and 'semantic' reasoning to be quite powerful.

In Section 9 it is shown how tactics can be formulated that correspond to reasoning based on verification conditions. This enables all HOL's infrastructure for goal directed proof to be brought to bear on partial correctness specifications. Before describing this, here is a quick review of tactics and tacticals abridged from [13]. Readers familiar with Cambridge LCF or HOL should skip this section.

8 Introduction to tactics and tacticals

A *tactic* is an ML function which is applied to a goal to reduce it to subgoals. A *tactical* is a (higher-order) ML function for combining tactics to build new tactics[14].

For example, if T_1 and T_2 are tactics, then the ML expression T_1 `THEN` T_2 evaluates to a tactic which first applies T_1 to a goal and then applies T_2 to each subgoal produced by T_1. The tactical `THEN` is an infixed ML function.

8.1 Tactics

It simplifies the description of tactics if the following ML type abbreviations are used:

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

If T is a tactic and g is a goal, then applying T to g (*i.e.* evaluating the ML expression $T\ g$) will result in an object of ML type `subgoals`, *i.e.* a pair whose first component is a list of goals and whose second component has ML type `proof`.

Suppose $T\ g$ = ([g_1;...;g_n],p). The idea is that g_1 , ... , g_n are subgoals and p is a 'validation' of the reduction of goal g to subgoals g_1 , ... , g_n. Suppose further that the subgoals g_1 , ... , g_n have been solved. This would mean that theorems th_1 , ... , th_n have been proved such that each th_i 'achieves' the goal g_i. The validation p (produced by applying T to g) is an ML function which when applied to the list [th_1;...;th_n] returns a theorem, th, which 'achieves' the original goal g. Thus p is a function for converting a solution of the subgoals to a solution of the original goal. If p does this successfully, then the tactic T is called *valid*. Invalid tactics cannot result

[14] The terms 'tactic' and 'tactical' are due to Robin Milner, who invented the concepts.

in the proof of invalid theorems; the worst they can do is result in insolvable goals or unintended theorems being proved. If T were invalid and were used to reduce goal g to subgoals g_1 , ... , g_n, then a lot of effort might be put into into proving theorems th_1 , ... , th_n achieving g_1 , ... , g_n, only to find that these theorems are useless because `p[th1;...;thn]` doesn't achieve g (*i.e.* it fails, or else it achieves some other goal).

A theorem *achieves* a goal if the assumptions of the theorem are included in the assumptions of the goal *and* if the conclusion of the theorem is equal (up to renaming of bound variables) to the conclusion of the goal. More precisely, a theorem t_1, ... , t_m `|-` t achieves a goal `([u1;...;un],u)` if and only if t_1, ... , t_m are included among u_1 , ... , u_n and t is equal to u (up to renaming of bound variables). For example, the goal `(["x=y";"y=z";"z=w"],"x=z")` is achieved by the theorem `x=y, y=z` ⊢ `x=z` (the assumption `"z=w"` is not needed).

A tactic *solves* a goal if it reduces the goal to the empty list of subgoals. Thus T solves g if `T g = ([],p)`. If this is the case, and if T is valid, then `p[]` will evaluate to a theorem achieving g. Thus if T solves g then the ML expression `snd(T g)[]` evaluates to a theorem achieving g.

Tactics are specified using the following notation:

$$\frac{goal}{goal_1 \quad goal_2 \quad \ldots \quad goal_n}$$

For example, the tactic `CONJ_TAC` is specified by

$$\frac{t_1 \ /\backslash \ t_2}{t_1 \qquad t_2}$$

`CONJ_TAC` reduces a goal of the form `(Γ,"t1/\t2")` to subgoals `(Γ,"t1")` and `(Γ,"t2")`. The fact that the assumptions of the top-level goal are propagated unchanged to the two subgoals is indicated by the absence of assumptions in the notation.

Tactics generally 'fail' (in the ML sense [13]) if they are applied to inappropriate goals. For example, `CONJ_TAC` will fail if it is applied to a goal whose conclusion is not a conjunction.

8.2 Using Tactics to Prove Theorems

Suppose a goal g is to be solved. If g is simple it might be possible to think up a tactic, T say, which reduces it to the empty list of subgoals. If this is the case then executing

```
let gl,p = T g
```

will bind p to a function which when applied to the empty list of theorems yields a theorem th achieving g. The declaration above will also bind gl to the empty list of goals. Executing

```
let th = p []
```

will thus bind th to a theorem achieving g.

To simplify the use of tactics, there is a standard function called `TAC_PROOF` with type `goal # tactic -> thm` such that evaluating

```
TAC_PROOF(g,T)
```

proves the goal g using tactic T and returns the resulting theorem (or fails, if T does not solve g).

When conducting a proof that involves many subgoals and tactics, it is necessary to keep track of all the validations and compose them in the correct order. While this is feasible even in large proofs, it is tedious. HOL provides a package for building and traversing the tree of subgoals, stacking the validations and applying them when subgoals are solved; this package was originally implemented for LCF by Larry Paulson [30].

The subgoal package implements a simple framework for interactive proof in which proof trees can be created and traversed in a top-down fashion. Using a tactic, the current goal is expanded into subgoals and a validation, which are automatically pushed onto the goal stack. Subgoals can be attacked in any order. If the tactic solves the goal (i.e. returns an empty subgoal list), then the package proceeds to the next goal in the tree.

The function `goal` has type `term -> void` and takes a term t and then sets up the goal `([],`t`)`.

The function `expand` has type `tactic -> void` and applies a tactic to the top goal on the stack, then pushes the resulting subgoals onto the stack, then prints the resulting subgoals. If there are no subgoals, the validation is applied to the theorems solving the subgoals that have been proved and the resulting theorems are printed.

8.3 Tacticals

A *tactical* is an ML function that returns a tactic (or tactics) as result. Tacticals may take various parameters; this is reflected in the various ML types that the built-in tacticals have. The tacticals used in this paper are:

`ORELSE : tactic -> tactic -> tactic`

> The tactical `ORELSE` is an ML infix. If T_1 and T_2 are tactics, then the ML expression T_1 `ORELSE` T_2 evaluates to a tactic which first tries T_1 and then if T_1 fails it tries T_2.

`THEN : tactic -> tactic -> tactic`

> The tactical `THEN` is an ML infix. If T_1 and T_2 are tactics, then the ML expression T_1 `THEN` T_2 evaluates to a tactic which first applies T_1 and then applies T_2 to all the subgoals produced by T_1.

`THENL : tactic -> tactic list -> tactic`

> If T is a tactic which produces n subgoals and $T_1, \ldots, T_n$ are tactics, then T `THENL [`T_1`;`...`;`T_n`]` is a tactic which first applies T and then applies T_i to the ith subgoal produced by T. The tactical `THENL` is useful for doing different things to different subgoals.

`REPEAT : tactic -> tactic`

> If T is a tactic then `REPEAT` T is a tactic which repeatedly applies T until it fails.

9 Verification conditions via tactics

If one wants to prove the partial correctness specification $\{\mathcal{P}\}\ \mathcal{V}\texttt{:=}\mathcal{E}\ \{\mathcal{Q}\}$ then, by the assignment axiom and precondition strengthening, it is clearly sufficient to prove the pure logic theorem $\vdash\ \mathcal{P}\ \Rightarrow\ \mathcal{Q}[\mathcal{E}/\mathcal{V}]$. The formula $\mathcal{P}\ \Rightarrow\ \mathcal{Q}[\mathcal{E}/\mathcal{V}]$ is called the *verification condition* for $\{\mathcal{P}\}\ \mathcal{V}\texttt{:=}\mathcal{E}\ \{\mathcal{Q}\}$.

More generally, the verification conditions for $\{\mathcal{P}\}\ \mathcal{C}\ \{\mathcal{Q}\}$ are a set of pure logic formulae $\mathcal{F}_1, \ldots, \mathcal{F}_n$ such that if $\vdash\ \mathcal{F}_1, \ldots, \vdash\ \mathcal{F}_n$ are theorems of pure logic then $\vdash\ \{\mathcal{P}\}\ \mathcal{C}\ \{\mathcal{Q}\}$ is a theorem of Hoare logic. Verification conditions are related to Dijkstra's weakest liberal preconditions [6], see the definition of Wlp in Section 11.2. For example, the weakest liberal precondition of $\mathcal{Q}$ for the assignment command $\mathcal{V}\texttt{:=}\ \mathcal{E}$ is $\mathcal{Q}[\mathcal{E}/\mathcal{V}]$. The verification conditions for $\{\mathcal{P}\}\ \mathcal{C}\ \{\mathcal{Q}\}$ is $\mathcal{P}\ \Rightarrow\ \mathsf{Wlp}(\mathcal{C}, \mathcal{Q})$ and it is possible that the treatment of verification conditions that follows could be improved by formulating it in terms of Wlp.

The generation of verification conditions can be represented by a tactic

$$\frac{\{\mathcal{P}\}\ \mathcal{C}\ \{\mathcal{Q}\}}{\mathcal{F}_1\quad \mathcal{F}_2\quad \ldots\quad \mathcal{F}_n}$$

where the validation (proof) part of the tactic is a composition of the functions representing the axioms and rules of Hoare logic. For example, the tactic `ASSIGN_TAC` is:

$$\frac{\{\mathcal{P}\}\ \mathcal{V}\texttt{:=}\ \mathcal{E}\ \{\mathcal{Q}\}}{\mathcal{P}\Rightarrow\mathcal{Q}[\mathcal{E}/\mathcal{V}]}$$

Here is a little session illustrating `ASSIGN_TAC`.

```
#goal "{X=x} X:=X+1 {X=x+1}" ;;
"{X = x} X := X + 1 {X = x + 1}"

#expand ASSIGN_TAC;;
OK..
"(X = x) ⇒ (X + 1 = x + 1)"

#expand  'Move antecedent to assumptions and then simplify' ;;
OK..
goal proved
⊢ (X = x) ⇒ (X + 1 = x + 1)
⊢ {X = x} X := X + 1 {X = x + 1}

Previous subproof:
goal proved
```

Each of the Hoare axioms and rules can be 'inverted' into a tactic which accomplishes one step in the process of verification condition generation. The composition of these tactics then results in a complete verification condition generator.

The specification of these tactics is as follows.

`SKIP_TAC : tactic`

$$\frac{\{\mathcal{P}\}\ \mathbf{skip}\ \{\mathcal{P}\}}{}$$

`SKIP_TAC` solves all goals of the form $\{\mathcal{P}\}$ **skip** $\{\mathcal{P}\}$; it generates an empty list of verification conditions.

`ASSIGN_TAC : tactic`

$$\frac{\{\mathcal{P}\}\ \mathcal{V}\texttt{:=}\ \mathcal{E}\ \{\mathcal{Q}\}}{\mathcal{P}\Rightarrow\mathcal{Q}[\mathcal{E}/\mathcal{V}]}$$

`SEQ_TAC : tactic`

To make generating verification conditions for sequences simple, it is convenient to require annotations to be inserted in sequences $\mathcal{C}_1$;$\mathcal{C}_2$ when $\mathcal{C}_2$ is *not* an assignment [14]. Such an annotation will be of the form `assert`$\{\mathcal{R}\}$, where $\mathcal{R}$ is a pure logic formula. For such sequences `SEQ_TAC` is as follows:

$$\frac{\{\mathcal{P}\}\ \mathcal{C}_1\texttt{;}\ \texttt{assert}\{\mathcal{R}\}\texttt{;}\ \mathcal{C}_2\ \{\mathcal{Q}\}}{\{\mathcal{P}\}\mathcal{C}_1\{\mathcal{R}\} \qquad \{\mathcal{R}\}\mathcal{C}_2\{\mathcal{Q}\}}$$

In the case that $\mathcal{C}_2$ is an assignment $\mathcal{V}$:=$\mathcal{E}$, the postcondition $\mathcal{Q}$ can be 'passed through' $\mathcal{C}_2$, using the assignment axiom, to automatically generate the assertion $\mathcal{Q}[\mathcal{E}/\mathcal{V}]$. Thus in this case `SEQ_TAC` simplifies to:

$$\frac{\{\mathcal{P}\}\ \mathcal{C}\texttt{;}\ \mathcal{V}\texttt{:=}\ \mathcal{E}\ \{\mathcal{Q}\}}{\{\mathcal{P}\}\ \mathcal{C}\ \{\mathcal{Q}[\mathcal{E}/\mathcal{V}]\}}$$

`IF_TAC : tactic`

$$\frac{\{\mathcal{P}\}\ \texttt{if}\ \mathcal{B}\ \texttt{then}\ \mathcal{C}_1\ \texttt{else}\ \mathcal{C}_2\ \{\mathcal{Q}\}}{\{\mathcal{P}\wedge\mathcal{B}\}\ \mathcal{C}_1\ \{\mathcal{Q}\} \qquad \{\mathcal{P}\wedge\neg\mathcal{B}\}\ \mathcal{C}_2\ \{\mathcal{Q}\}}$$

`WHILE_TAC : tactic`

In order for verification conditions to be generated from **while**-commands, it is necessary to specify an invariant [14] by requiring that an *annotation* `invariant`$\{\mathcal{R}\}$, where $\mathcal{R}$ is the invariant, be inserted just after the `do`.

$$\frac{\{\mathcal{P}\}\ \texttt{while}\ \mathcal{B}\ \texttt{do}\ \texttt{invariant}\ \{\mathcal{R}\}\texttt{;}\ \mathcal{C}\ \{\mathcal{Q}\}}{\mathcal{P}\Rightarrow\mathcal{R} \qquad \{\mathcal{R}\wedge\mathcal{B}\}\ \mathcal{C}\ \ \{\mathcal{R}\} \qquad \mathcal{R}\wedge\neg\mathcal{B}\Rightarrow\mathcal{Q}}$$

Here now is a continuation of the session started above in which tactics for Hoare logic are illustrated. First the individual tactics are illustrated, then it is shown how they can be combined, using tacticals, into a single verification condition generator.

The first step is to set up the specification of the simple division program as a goal.

```
#pretty_on();;
() : void

#goal "{T}
#        R:=X;
#        Q:=0;
#        assert{(R = X) ∧ (Q = 0)};
#        while Y ≤ R
#         do (invariant{X = (R + (Y × Q))};
#             R := R − Y; Q := Q + 1)
#      {(X = (R + (Y × Q))) ∧ (R < Y)}" ;;
"{T}
  R := X;
  Q := 0;
  assert{(R = X) ∧ (Q = 0)};
  while Y ≤ R do invariant{X = R + (Y × Q)}; R := R − Y; Q := Q + 1
 {(X = R + (Y × Q)) ∧ R < Y}"
```

The command in this goal is a sequence, so `SEQ_TAC` can be applied.

```
#expand SEQ_TAC;;
OK..
2 subgoals
"{(R = X) ∧ (Q = 0)}
  while Y ≤ R do invariant{X = R + (Y × Q)};
   R := R − Y; Q := Q + 1
 {(X = R + (Y × Q)) ∧ R < Y}"

"{T} R := X; Q := 0 {(R = X) ∧ (Q = 0)}"
```

The top goal is printed last; `SEQ_TAC` can be applied to it.

```
#expand SEQ_TAC;;
OK..
"{T} R := X {(R = X) ∧ (0 = 0)}"

#expand ASSIGN_TAC;;
OK..
"T ⇒ (X = X) ∧ (0 = 0)"
```

The goal `T⇒(X =X) ∧(0 =0)` is solved by rewriting with standard facts. The subgoal package prints out the theorems produced and backs up to the next pending subgoal.

```
#expand ‘Simplify’ ;;
OK..
goal proved
⊢ T ⇒ (X = X) ∧ (0 = 0)
⊢ {T} R := X {(R = X) ∧ (0 = 0)}
⊢ {T} R := X; Q := 0 {(R = X) ∧ (Q = 0)}

Previous subproof:
"{(R = X) ∧ (Q = 0)}
  while Y ≤ R do invariant{X = R + (Y × Q)};
   R := R − Y; Q := Q + 1
 {R < Y ∧ (X = R + (Y × Q))}"
```

The remaining subgoal is a **while**-command, so `WHILE_TAC` is applied to get three subgoals.

```
#expand WHILE_TAC;;
OK..
3 subgoals
"(X = R + (Y × Q)) ∧ ¬ Y ≤ R ⇒ (X = R + (Y × Q)) ∧ R < Y"

"{(X = R + (Y × Q)) ∧ Y ≤ R}
  R := R − Y; Q := Q + 1
 {X = R + (Y × Q)}"

"(R = X) ∧ (Q = 0) ⇒ (X = R + (Y × Q))"
```

The first subgoal (i.e. the one printed last) is quickly solved by moving the antecedent of the implication to the assumptions, and then rewriting using standard properties of $+$ and $\times$.

```
#expand  ‘Move antecedent to assumptions’ ;;
OK..
"X = R + (Y × Q)"
     [ "R = X" ]
     [ "Q = 0" ]

#expand  ‘Simplify using assumptions’ ;;
OK..
goal proved
.. ⊢ X = R + (Y × Q)
⊢ (R = X) ∧ (Q = 0) ⇒ (X = R + (Y × Q))

Previous subproof:
2 subgoals
"(X = R + (Y × Q)) ∧ ¬ Y ≤ R ⇒ (X = R + (Y × Q)) ∧ R < Y"

"{(X = R + (Y × Q)) ∧ Y ≤ R}
  R := R − Y; Q := Q + 1
 {X = R + (Y × Q)}"
```

The first subgoal is a sequence, so `SEQ_TAC` is applied. This results in an assignment, so `ASSIGN_TAC` is then applied. The resulting purely logical subgoal is the already proved `Th7`.

```
#expand SEQ_TAC;;
OK..
"{(X = R + (Y × Q)) ∧ Y ≤ R} R := R − Y {X = R + (Y × (Q + 1))}"

#expand ASSIGN_TAC;;
OK..
"(X = R + (Y × Q)) ∧ Y ≤ R ⇒ (X = (R − Y) + (Y × (Q + 1)))"
```

`Th7` is used to solve the first subgoal.

```
#expand  ‘Use Th7 ’ ;;
OK..
goal proved
⊢ (X = R + (Y × Q)) ∧ Y ≤ R ⇒ (X = (R − Y) + (Y × (Q + 1)))
⊢ {(X = R + (Y × Q)) ∧ Y ≤ R}
    R := R − Y
  {X = R + (Y × (Q + 1))}
⊢ {(X = R + (Y × Q)) ∧ Y ≤ R}
    R := R − Y; Q := Q + 1
  {X = R + (Y × Q)}

Previous subproof:
"(X = R + (Y × Q)) ∧ ¬ Y ≤ R ⇒ (X = R + (Y × Q)) ∧ R < Y"
```

The previously proved theorem `Th12` solves the remaining subgoal.

```
#expand  ‘Use Th12 ’ ;;
OK..
goal proved
⊢ (X = R + (Y × Q)) ∧ ¬ Y ≤ R ⇒ (X = R + (Y × Q)) ∧ R < Y
⊢ {(R = X) ∧ (Q = 0)}
    while Y ≤ R do R := R − Y; Q := Q + 1
  {(X = R + (Y × Q)) ∧ R < Y}
⊢ {T}
    R := X; Q := 0; while Y ≤ R do R := R − Y; Q := Q + 1
  {(X = R + (Y × Q)) ∧ R < Y}

Previous subproof:
goal proved
```

The tactics `ASSIGN_TAC`, `SEQ_TAC`, `IF_TAC` and `WHILE_TAC` can be combined into a single tactic, called `VC_TAC` below, which generates the verification conditions in a single step.

The definition of `VC_TAC` in ML is given in the next box; it uses the tacticals `REPEAT` and `ORELSE` described in Section 8.3.

```
#let VC_TAC = REPEAT(ASSIGN_TAC
#                     ORELSE SEQ_TAC
#                     ORELSE IF_TAC
#                     ORELSE WHILE_TAC);;
VC_TAC = - : tactic
```

This compound tactic is illustrated by using it to repeat the proof just done. The original goal is made the top goal, and then this is expanded using `VC_TAC`. The result is four verification conditions.

```
#goal "{T}
#        R:=X;
#        Q:=0;
#        assert{(R = X) ∧ (Q = 0)};
#        while Y ≤ R
#         do (invariant{X = (R + (Y × Q))};
#             R := R − Y; Q := Q + 1)
#      {(X = (R + (Y × Q))) ∧ (R < Y)}" ;;
"{T}
  R := X;
  Q := 0;
  assert{(R = X) ∧ (Q = 0)};
  while Y ≤ R do invariant{X = R + (Y × Q)};
   R := R − Y; Q := Q + 1
 {(X = R + (Y × Q)) ∧ R < Y}"

#expand VC_TAC;;
OK..
4 subgoals
"(X = R + (Y × Q)) ∧ ¬ Y ≤ R ⇒ (X = R + (Y × Q)) ∧ R < Y"

"(X = R + (Y × Q)) ∧ Y ≤ R ⇒ (X = (R − Y) + (Y × (Q + 1)))"

"(R = X) ∧ (Q = 0) ⇒ (X = R + (Y × Q))"

"T ⇒ (X = X) ∧ (0 = 0)"
```

Notice how `VC_TAC` converts the goal of proving a Hoare specification into pure logic subgoals. These can be solved as above.

```
#expand  'Simplify' ;;
OK..
goal proved
⊢ T ⇒ (X = X) ∧ (0 = 0)

Previous subproof:
3 subgoals
"(X = R + (Y × Q)) ∧ ¬ Y ≤ R ⇒ (X = R + (Y × Q)) ∧ R < Y"

"(X = R + (Y × Q)) ∧ Y ≤ R ⇒ (X = (R − Y) + (Y × (Q + 1)))"

"(R = X) ∧ (Q = 0) ⇒ (X = R + (Y × Q))"
```

```
#expand  'Move antecedent to assumptions and then rewrite' ;;
OK..
goal proved
⊢ (R = X) ∧ (Q = 0) ⇒ (X = R + (Y × Q))

Previous subproof:
2 subgoals
"(X = R + (Y × Q)) ∧ ¬ Y ≤ R ⇒ (X = R + (Y × Q)) ∧ R < Y"

"(X = R + (Y × Q)) ∧ Y ≤ R ⇒ (X = (R − Y) + (Y × (Q + 1)))"
```

```
#expand ‘Use Th7 ’ ;;
OK..
goal proved
⊢ (X = R + (Y × Q)) ∧ Y ≤ R ⇒ (X = (R − Y) + (Y × (Q + 1)))

Previous subproof:
"(X = R + (Y × Q)) ∧ ¬ Y ≤ R ⇒ (X = R + (Y × Q)) ∧ R < Y"
```

```
#expand ‘Use Th12 ’ ;;
OK..
goal proved
⊢ (X = R + (Y × Q)) ∧ ¬ Y ≤ R ⇒ (X = R + (Y × Q)) ∧ R < Y
⊢ {T}
    R := X; Q := 0; while Y ≤ R do R := R − Y; Q := Q + 1
  {(X = R + (Y × Q)) ∧ R < Y}

Previous subproof:
goal proved
```

Finally, here is the proof in one step.

```
#prove
# ("{T},
#       R:=X;
#       Q:=0;
#       assert{(R = X) ∧ (Q = 0)};
#       while Y ≤ R
#        do (invariant{X = (R + (Y × Q))};
#            R:=R − Y; Q:=Q + 1)
#   {(X = (R + (Y × Q))) ∧ R < Y}",
#   VC_TAC
#    THENL
#     [ ‘Simplify’ ;
#       ‘Move antecedent to assumptions and then simplify’ ;
#       ‘Use Th7 ’ ;
#       ‘Use Th12 ’ ]);;
⊢ {T}
    R := X; Q := 0; while Y ≤ R do R := R − Y; Q := Q + 1
  {(X = R + (Y × Q)) ∧ R < Y}
```

10 Termination and total correctness

Hoare logic is usually presented as a self-contained calculus. However, if it is regarded as a derived logic, as it is here, then it's easy to add extensions and modifications without fear of introducing unsoundness. To illustrate this, we will sketch how termination assertions can be added, and how these can be used to prove total correctness.

A *termination assertion* is a formula $\mathsf{Halts}([\![\mathcal{P}]\!], [\![\mathcal{C}]\!])$, where the constant Halts is defined by:

$$\mathsf{Halts}(p, c) \;=\; \forall s_1.\ p\ s_1 \;\Rightarrow\; \exists s_2.\ c(s_1, s_2)$$

Notice that this says that c 'halts' under precondition p if there is *some* final state for each initial state satisfying p. For example, although **while** T **do skip** does not

terminate, the definition above suggests that (**while** T **do skip**) $\|$ **skip** does, since:

$$\vdash\ \mathsf{Halts}([\![\mathsf{T}]\!],\ \mathsf{Choose}([\![\textbf{while}\ \mathsf{T}\ \textbf{do skip}]\!],\ [\![\textbf{skip}]\!]))$$

($\|$ and Choose are described in Section 5). The meaning of $\mathsf{Halts}([\![\mathcal{P}]\!], [\![\mathcal{C}]\!])$ is 'some computation of $\mathcal{C}$ starting from a state satisfying $\mathcal{P}$ terminates' this is quite different from 'every computation of $\mathcal{C}$ starting from a state satisfying $\mathcal{P}$ terminates'. The latter stronger kind of termination requires a more complex kind of semantics for its formalization (e.g. one using powerdomains [31]). If commands are deterministic, then termination is adequately formalized by Halts. It is intuitively clear (and can be proved using the methods described in Melham's paper [25]) that the relations denoted by commands in our little language (*not* including $\|$) are partial functions. If Det is defined by:

$$\mathsf{Det}\ c\ =\ \forall s\ s_1\ s_2.\ c(s,s_1)\ \wedge\ c(s,s_2)\ \Rightarrow\ (s_1 = s_2)$$

then for any command $\mathcal{C}$ it can be proved that $\vdash\ \mathsf{Det}\ [\![\mathcal{C}]\!]$. This fact will be needed to show that the formalization of weakest preconditions in Section 11.2 is correct.

The informal equation

$$\text{Total correctness} = \text{Termination} + \text{Partial correctness.}$$

can be implemented by defining:

$$\mathsf{Total_Spec}(p,c,q)\ =\ \mathsf{Halts}(p,c)\ \wedge\ \mathsf{Spec}(p,c,q)$$

Then $[\mathcal{P}]\ \mathcal{C}\ [\mathcal{Q}]$ is represented by $\mathsf{Total_Spec}([\![\mathcal{P}]\!], [\![\mathcal{C}]\!], [\![\mathcal{Q}]\!])$.

From the definition of Halts it is straightforward to prove the following theorems:

T1. $\vdash\ \forall p.\ \mathsf{Halts}(p, \mathsf{Skip})$

T2. $\vdash\ \forall p\ v\ e.\ \mathsf{Halts}(p, \mathsf{Assign}(v,e))$

T3. $\vdash\ \forall p\ p'\ c.\ (\forall s.\ p'\ s\ \Rightarrow\ p\ s)\ \wedge\ \mathsf{Halts}(p,c)\ \Rightarrow\ \mathsf{Halts}(p',c)$

T4. $\vdash\ \forall p\ c_1\ c_2\ q.\ \mathsf{Halts}(p,c_1)\ \wedge\ \mathsf{Spec}(p,c_1,q)\ \wedge\ \mathsf{Halts}(q,c_2)$
$\Rightarrow \mathsf{Halts}(p, \mathsf{Seq}(c_1,c_2))$

T5. $\vdash\ \forall p\ c_1\ c_2\ b.\ \mathsf{Halts}(p,c_1)\ \wedge\ \mathsf{Halts}(p,c_2)\ \Rightarrow\ \mathsf{Halts}(p, \mathsf{If}(b,c_1,c_2))$

T6. $\vdash\ \forall b\ c\ x.$
$(\forall n.\ \mathsf{Spec}((\lambda s.\ p\ s\ \wedge\ b\ s\ \wedge\ (s\ x = n)), c, (\lambda s.\ p\ s\ \wedge\ s\ x < n)))$
$\wedge\ \mathsf{Halts}((\lambda s.\ p\ s\ \wedge\ b\ s), c)$
$\Rightarrow \mathsf{Halts}(p, \mathsf{While}(b,c))$

Although these theorems are fairly obvious, when I first wrote them down I got a few details wrong. These errors soon emerged when the proofs were done using the HOL system.

T6 shows that if x is a *variant*, i.e. a variable whose value decreases each time 'around the loop', then the **while**-command halts. Proving this in HOL was much harder than any of the other theorems (but was still essentially routine).

10.1 Derived rules for total correctness

Using T1 – T6 above and H1 – H7 of Section 6, it is straightforward to apply the methods described in Section 6 to implement the derived rules for total correctness shown below. These are identical to the corresponding rules for partial correctness except for having '[' and ']' instead of '{' and '}' respectively.

$$\vdash\ [\mathcal{P}]\ \textbf{skip}\ [\mathcal{P}]$$

$$\frac{\vdash\ \mathcal{P}'\ \Rightarrow\ \mathcal{P} \qquad \vdash\ [\mathcal{P}]\ \mathcal{C}\ [\mathcal{Q}]}{\vdash\ [\mathcal{P}']\ \mathcal{C}\ [\mathcal{Q}]}$$

$$\frac{\vdash\ [\mathcal{P}]\ \mathcal{C}\ [\mathcal{Q}] \qquad \vdash\ \mathcal{Q}\ \Rightarrow\ \mathcal{Q}'}{\vdash\ [\mathcal{P}]\ \mathcal{C}\ [\mathcal{Q}']}$$

$$\frac{\vdash\ [\mathcal{P}]\ \mathcal{C}_1\ [\mathcal{Q}] \qquad \vdash\ [\mathcal{Q}]\ \mathcal{C}_2\ [\mathcal{R}]}{\vdash\ [\mathcal{P}]\ \mathcal{C}_1;\ \mathcal{C}_2\ [\mathcal{R}]}$$

$$\frac{\vdash\ [\mathcal{P}\ \wedge\ \mathcal{B}]\ \mathcal{C}_1\ [\mathcal{Q}] \qquad \vdash\ [\mathcal{P}\ \wedge\ \neg\mathcal{B}]\ \mathcal{C}_2\ [\mathcal{Q}]}{\vdash\ [\mathcal{P}]\ \textbf{if}\ \mathcal{B}\ \textbf{then}\ \mathcal{C}_1\ \textbf{else}\ \mathcal{C}_2\ [\mathcal{Q}]}$$

The total correctness rule for **while**-commands needs a stronger hypothesis than the corresponding one for partial correctness. This is to ensure that the command terminates. For this purpose, a variant is needed in addition to an invariant.

$$\frac{\vdash\ [\mathcal{P}\ \wedge\ \mathcal{B}\ \wedge\ (\mathtt{N} = n)]\ \mathcal{C}\ [\mathcal{P}\ \wedge\ (\mathtt{N} < n)]}{\vdash\ [\mathcal{P}]\ \textbf{while}\ \mathcal{B}\ \textbf{do}\ \mathcal{C}\ [\mathcal{P}\ \wedge\ \neg\mathcal{B}]}$$

Notice that since

$$\mathsf{Total_Spec}(p, c, q)\ =\ \mathsf{Halts}(p, c)\ \wedge\ \mathsf{Spec}(p, c, q)$$

it is clear that the following rule is valid

$$\frac{\vdash\ [\mathcal{P}]\ \mathcal{C}\ [\mathcal{Q}]}{\vdash\ \{\mathcal{P}\}\ \mathcal{C}\ \{\mathcal{Q}\}}$$

The converse to this is only valid if $\mathcal{C}$ contains no **while**-commands. It would be straightforward to implement a HOL derived rule

$$\frac{\vdash\ \{\mathcal{P}\}\ \mathcal{C}\ \{\mathcal{Q}\}}{\vdash\ [\mathcal{P}]\ \mathcal{C}\ [\mathcal{Q}]}$$

that would fail (in the ML sense) if $\mathcal{C}$ contained **while**-commands.

10.2 Tactics for total correctness

Tactics for total correctness can be implemented that use the derived rules in the previous section as validations. The tactics for everything except **while**-commands are obtained by replacing '[' and ']' by '{' and '}'. Namely:

SKIP_T_TAC : tactic

$$\frac{}{[\mathcal{P}]\ \textbf{skip}\ [\mathcal{P}]}$$

ASSIGN_T_TAC : tactic

$$\frac{[\mathcal{P}]\ \mathcal{V}\texttt{:=}\ \mathcal{E}\ [\mathcal{Q}]}{\mathcal{P}\Rightarrow\mathcal{Q}[\mathcal{E}/\mathcal{V}]}$$

SEQ_T_TAC : tactic

$$\frac{[\mathcal{P}]\ \mathcal{C}_1\texttt{;}\ \texttt{assert}\{\mathcal{R}\}\texttt{;}\ \mathcal{C}_2\ [\mathcal{Q}]}{[\mathcal{P}]\mathcal{C}_1[\mathcal{R}] \qquad [\mathcal{R}]\mathcal{C}_2[\mathcal{Q}]}$$

$$\frac{[\mathcal{P}]\ \mathcal{C}\texttt{;}\ \mathcal{V}\texttt{:=}\ \mathcal{E}\ [\mathcal{Q}]}{[\mathcal{P}]\ \mathcal{C}\ [\mathcal{Q}[\mathcal{E}/\mathcal{V}]]}$$

IF_T_TAC : tactic

$$\frac{[\mathcal{P}]\ \texttt{if}\ \mathcal{B}\ \texttt{then}\ \mathcal{C}_1\ \texttt{else}\ \mathcal{C}_2\ [\mathcal{Q}]}{[\mathcal{P}\ \wedge\mathcal{B}]\ \mathcal{C}_1\ [\mathcal{Q}] \qquad [\mathcal{P}\ \wedge\neg\mathcal{B}]\ \mathcal{C}_2\ [\mathcal{Q}]}$$

WHILE_T_TAC : tactic

> To enable verification conditions to be generated from **while**-commands they must be annotated with a variant as well as an invariant.

$$\frac{[\mathcal{P}]\ \texttt{while}\ \mathcal{B}\ \texttt{do}\ \texttt{invariant}\ \{\mathcal{R}\}\texttt{;}\ \texttt{variant}\ \{\texttt{N}\}\texttt{;}\ \mathcal{C}\ [\mathcal{Q}]}{\mathcal{P}\Rightarrow\mathcal{R} \qquad [\mathcal{R}\wedge\mathcal{B}\wedge(\texttt{N}=n)]\ \mathcal{C}\ \ [\mathcal{R}\wedge(\texttt{N}<n)] \qquad \mathcal{R}\wedge\neg\mathcal{B}\Rightarrow\mathcal{Q}}$$

To illustrate these tactics, here is a session in which the total correctness of the division program is proved. First suppose that a verification condition generator is defined by:

```
#let VC_T_TAC =
# REPEAT(ASSIGN_T_TAC
#        ORELSE SEQ_T_TAC
#        ORELSE IF_T_TAC
#        ORELSE WHILE_T_TAC);;
VC_T_TAC = - : tactic
```

and the following goal is set up:

```
##pretty_on();;
() : void

#goal "[0 < Y]
#       R := X;
#       Q := 0;
#       assert{(0 < Y) ∧ (R = X) ∧ (Q = 0)};
#       while Y ≤ R
#        do (invariant{(0 < Y) ∧ (X = R + (Y × Q))}; variant{R};
#        R := R - Y; Q := Q + 1)
#     [(X = R + (Y × Q)) ∧ R < Y]" ;;
"[0 < Y]
  R := X;
  Q := 0;
  assert{0 < Y ∧ (R = X) ∧ (Q = 0)};
  while Y ≤ R do
   invariant{0 < Y ∧ (X = R + (Y × Q))};
   variant{R};
   R := R - Y; Q := Q + 1
 [(X = R + (Y × Q)) ∧ R < Y]"

() : void
```

then applying VC_T_GEN results in the following four verification conditions:

```
#expand VC_T_TAC;;
OK..
4 subgoals
"(0 < Y ∧ (X = R + (Y × Q))) ∧ ¬ Y ≤ R ⇒
 (X = R + (Y × Q)) ∧ R < Y"

"(0 < Y ∧ (X = R + (Y × Q))) ∧ Y ≤ R ∧ (R = r) ⇒
 (0 < Y ∧ (X = (R - Y) + (Y × (Q + 1)))) ∧ (R - Y) < r"

"0 < Y ∧ (R = X) ∧ (Q = 0) ⇒ 0 < Y ∧ (X = R + (Y × Q))"

"0 < Y ⇒ 0 < Y ∧ (X = X) ∧ (0 = 0)"

() : void
```

These are routine to prove using HOL. Notice that WHILE_T_TAC has been implemented so that it automatically generates a logical (or ghost) variable by lowering the case of the variant. In the example above, r is a logical variable generated from the program variable R that is given as the variant.

11 Other programming logic constructs

In this section, three variants on Hoare logic are described.

(i) VDM-style specifications.

(ii) Weakest preconditions.

(iii) Dynamic logic.

None have these have been fully mechanized in HOL, but it is hoped that enough detail is given to show that doing so should be straightforward.

11.1 VDM-style specifications

The Vienna Development Method (VDM) [19]) is a formal method for program development which uses a variation on Hoare-style specifications. The VDM notation reduces the need for auxiliary logical variables by providing a way of refering to the initial values of variables in postconditions. For example, the following Hoare-style partial correctness specification:

$$\{X = x \ \wedge \ Y = y\} \ R := X;\ X := Y;\ Y := R \ \{Y = x \ \wedge \ X = y\}$$

could be written in a VDM-style as:

$$\{\mathsf{T}\} \ R := X;\ X := Y;\ Y := R \ \{Y = \overleftarrow{X} \ \wedge \ X = \overleftarrow{Y}\}$$

where $\overleftarrow{X}$ and $\overleftarrow{Y}$ denote the values X and Y had before the three assignments were executed. More generally,

$$\{\mathcal{P}[X_1, \ldots, X_n]\} \ \mathcal{C} \ \{\mathcal{Q}[X_1, \ldots, X_n, \overleftarrow{X_1}, \ldots, \overleftarrow{X_n}]\}$$

can be thought of as an abbreviation for

$$\{\mathcal{P}[X_1, \ldots, X_n] \ \wedge \ X_1 = \overleftarrow{X_1} \ \wedge \ \ldots \ \wedge \ X_n = \overleftarrow{X_n}\} \ \mathcal{C} \ \{\mathcal{Q}[X_1, \ldots, X_n, \overleftarrow{X_1}, \ldots, \overleftarrow{X_n}]\}$$

where $\overleftarrow{X_1}$, ... , $\overleftarrow{X_n}$ are distinct logical variables not occurring in $\mathcal{C}$. It should be straightforward to build a parser and pretty-printer that supports this interpretation of VDM specifications.

It is claimed that VDM specifications are more natural than convential Hoare-style ones. I have not worked with them enough to have an opinion on this, but the point I hope to make here is that there is no problem mechanizing a VDM-style programming logic using the methods in this paper.

Although the meaning of individual VDM specifications is clear, it is not so easy to see what the correct Hoare-like rules of inference are. For example, the sequencing rule must somehow support the deduction of

$$\{\mathsf{T}\} \ X := X + 1;\ X := X + 1 \ \{X = \overleftarrow{X} + 2\}$$

from

$$\{\mathsf{T}\} \ X := X + 1 \ \{X = \overleftarrow{X} + 1\}$$

There is another semantics of VDM specifications, which Jones attributes to Peter Aczel [19]. This semantics avoids the need for hidden logical variables and also makes it easy to see what the correct rules of inference are. The idea is to regard the post-condition as a binary relation on the initial and final states. This can be formalized by regarding

$$\{\mathcal{P}[X_1, \ldots, X_n]\} \ \mathcal{C} \ \{\mathcal{Q}[X_1, \ldots, X_n, \overleftarrow{X_1}, \ldots, \overleftarrow{X_n}]\}$$

as an abbreviation for

$$\mathsf{VDM_Spec}([\![\mathcal{P}[X_1, \ldots, X_n]]\!],\ [\![\mathcal{C}]\!],\ [\![\mathcal{Q}[X_1, \ldots, X_n, \overleftarrow{X_1}, \ldots, \overleftarrow{X_n}]]\!]_2)$$

where $\mathsf{VDM_Spec}$ is defined by:

$$\mathsf{VDM_Spec}(p, c, r) = \forall s_1\ s_2.\ p\ s_1\ \wedge\ c(s_1, s_2)\ \Rightarrow\ r(s_1, s_2)$$

and the notation $[\![\ \cdots\ ]\!]_2$ is defined by:

$$\begin{array}{l} [\![\mathcal{Q}[X_1, \ldots, X_n, \overleftarrow{X_1}, \ldots, \overleftarrow{X_n}]]\!]_2\ = \\ \quad \lambda(s_1, s_2).\ \mathcal{Q}[s_2\text{`}X_1\text{`}, \ldots, s_2\text{`}X_n\text{`}, s_1\text{`}X_1\text{`}, \ldots, s_1\text{`}X_n\text{`}] \end{array}$$

It is clear that $[\![\ \cdots\ ]\!]_2$ could be supported by a parser and pretty-printer in the same way that $[\![\ \cdots\ ]\!]$ is supported.

The sequencing rule now corresponds to the theorem:

$$\begin{array}{l} \vdash\ \forall p_1\ p_2\ r_1\ r_2\ c_1\ c_2. \\ \quad \mathsf{VDM_Spec}(p_1,\ c_1,\ \lambda(s_1, s_2).\ p_2\ s_2\ \wedge\ r_1(s_1, s_2))\ \wedge \\ \quad \mathsf{VDM_Spec}(p_2,\ c_2,\ r_2)\ \Rightarrow \\ \quad \mathsf{VDM_Spec}(p_1,\ \mathsf{Seq}(c_1, c_2),\ \mathsf{Seq}(r_1, r_2)) \end{array}$$

Example

If $\{\mathsf{T}\}\ X := X + 1\ \{X = \overleftarrow{X} + 1\}$ is interpreted as:

$$\mathsf{VDM_Spec}([\![\mathsf{T}]\!],\ [\![X := X + 1]\!],\ [\![X = \overleftarrow{X} + 1]\!]_2)$$

which (since $\vdash\ \forall x.\ \mathsf{T}\ \wedge\ x\ =\ x$) implies:

$$\mathsf{VDM_Spec}([\![\mathsf{T}]\!],\ [\![X := X + 1]\!],\ \lambda(s_1, s_2).\ [\![\mathsf{T}]\!]s_2\ \wedge\ [\![X = \overleftarrow{X} + 1]\!]_2(s_1, s_2))$$

and hence it follows by the sequencing theorem above that:

$$\mathsf{VDM_Spec}([\![\mathsf{T}]\!],\ [\![X := X + 1;\ X := X + 1]\!],\ \mathsf{Seq}([\![X = \overleftarrow{X} + 1]\!]_2,\ [\![X = \overleftarrow{X} + 1]\!]_2))$$

By the definition of Seq in Section 5:

$$\begin{array}{l} \mathsf{Seq}([\![X = \overleftarrow{X} + 1]\!]_2,\ [\![X = \overleftarrow{X} + 1]\!]_2)(s_1, s_2) \\ \quad =\ \exists s.\ [\![X = \overleftarrow{X} + 1]\!]_2(s_1, s)\ \wedge\ [\![X = \overleftarrow{X} + 1]\!]_2(s, s_2) \\ \quad =\ \exists s.\ (\lambda(s_1, s_2).\ s_2\text{`}X\text{`} = s_1\text{`}X\text{`} + 1)(s_1, s)\ \wedge\ (\lambda(s_1, s_2).\ s_2\text{`}X\text{`} = s_1\text{`}X\text{`} + 1)(s, s_2) \\ \quad =\ \exists s.\ (s\text{`}X\text{`} = s_1\text{`}X\text{`} + 1)\ \wedge\ (s_2\text{`}X\text{`} = s\text{`}X\text{`} + 1) \\ \quad =\ \exists s.\ (s\text{`}X\text{`} = s_1\text{`}X\text{`} + 1)\ \wedge\ (s_2\text{`}X\text{`} = (s_1\text{`}X\text{`} + 1) + 1) \\ \quad =\ \exists s.\ (s\text{`}X\text{`} = s_1\text{`}X\text{`} + 1)\ \wedge\ (s_2\text{`}X\text{`} = s_1\text{`}X\text{`} + 2) \\ \quad =\ (\exists s.\ s\text{`}X\text{`} = s_1\text{`}X\text{`} + 1)\ \wedge\ (\exists s.\ s_2\text{`}X\text{`} = s_1\text{`}X\text{`} + 2) \\ \quad =\ \mathsf{T}\ \ \wedge\ (s_2\text{`}X\text{`} = s_1\text{`}X\text{`} + 2) \\ \quad =\ (s_2\text{`}X\text{`} = s_1\text{`}X\text{`} + 2) \\ \quad =\ [\![X = \overleftarrow{X} + 2]\!]_2(s_1, s_2) \end{array}$$

Hence:

$$\vdash\ \{\mathsf{T}\}\ X := X + 1;\ X := X + 1\ \{X = \overleftarrow{X} + 2\}$$

□

An elegant application of treating postconditions as binary relations is Aczel's version of the **while**-rule [19]:

$$\frac{\vdash\ \{\mathcal{P}\ \wedge\ \mathcal{B}\}\ \mathcal{C}\ \{\mathcal{P}\ \wedge\ \mathcal{R}\}}{\vdash\ \{\mathcal{P}\}\ \textbf{while}\ \mathcal{B}\ \textbf{do}\ \mathcal{C}\ \{\mathcal{P}\ \wedge\ \neg\mathcal{B}\ \wedge\ \mathcal{R}^*\}}$$

Where $\mathcal{R}^*$ is the reflexive closure of $\mathcal{R}$ defined by

$$\mathcal{R}^*(s_1, s_2)\ =\ \exists n.\ \mathcal{R}^n(s_1, s_2)$$

and $\mathcal{R}^n$ is definable in higher order logic by the following primitive recursion:

$$\begin{array}{lcl} \mathcal{R}^0 & = & \lambda(s_1, s_2).\ (s_1 = s_2) \\ \mathcal{R}^{n+1} & = & \mathsf{Seq}(\mathcal{R}, \mathcal{R}^n) \end{array}$$

Aczel pointed out that his version of the **while**-rule can be converted into a rule of total correctness simply by requiring $\mathcal{R}$ to be transitive and well-founded:

$$\frac{\vdash\ [\mathcal{P}\ \wedge\ \mathcal{B}]\ \mathcal{C}\ [\mathcal{P}\ \wedge\ \mathcal{R}] \qquad \vdash\ \mathsf{Transitive}\ \mathcal{R} \qquad \vdash\ \mathsf{Well_Founded}\ \mathcal{R}}{\vdash\ [\mathcal{P}]\ \textbf{while}\ \mathcal{B}\ \textbf{do}\ \mathcal{C}\ [\mathcal{P}\ \wedge\ \neg\mathcal{B}\ \wedge\ \mathcal{R}^*]}$$

where:

$$\begin{array}{lcl} \mathsf{Transitive}\ r & = & \forall s_1\ s_2\ s_3.\ r(s_1, s_2)\ \wedge\ r(s_2, s_3)\ \Rightarrow\ r(s_1, s_3) \\ \mathsf{Well_Founded}\ r & = & \neg\exists f : \mathit{num}{\rightarrow}\mathit{state}.\ \forall n.\ r(f(n), f(n+1)) \end{array}$$

Notice how it is straightforward to define notions like Transitive and Well_Founded in higher order logic; these cannot be defined in first order logic.

11.2 Dijkstra's weakest preconditions

Dijkstra's theory of weakest preconditions, like VDM, is primarily a theory of rigorous program construction rather than a theory of post hoc verification. As will be shown, it is straightforward to define weakest preconditions for deterministic programs in higher order logic[15].

In his book [6], Dijkstra introduced both 'weakest liberal preconditions' (Wlp) and 'weakest preconditions' (Wp); the former for partial correctness and the latter for total correctness. The idea is that if $\mathcal{C}$ is a command and $\mathcal{Q}$ a predicate, then:

- $\mathsf{Wlp}(\mathcal{C}, \mathcal{Q})\ =$ '*The weakest predicate* $\mathcal{P}$ *such that* $\{\mathcal{P}\}\ \mathcal{Q}\ \{\mathcal{Q}\}$'
- $\mathsf{Wp}(\mathcal{C}, \mathcal{Q})\ =$ '*The weakest predicate* $\mathcal{P}$ *such that* $[\mathcal{P}]\ \mathcal{Q}\ [\mathcal{Q}]$'

Before defining these notions formally, it is necessary to first define the general notion of the 'weakest predicate' satisfying a condition. If p and q are predicates on states (i.e. have type $\mathit{state}{\rightarrow}\mathit{bool}$), then define $p{\Leftarrow}q$ to mean p is weaker (i.e. 'less constraining') than q, in the sense that everything satisfying q also satisfies p. Formally:

$$p{\Leftarrow}q\ =\ \forall s.\ q\ s\ \Rightarrow\ p\ s$$

The weakest predicate satisfying a condition can be given a general definition using Hilbert's ε-operator. This is an operator that chooses an object satisfying a predicate. If P is a predicate on predicates, then $\varepsilon p.\ P\ p$ is the predicate defined by the property:

$$(\exists p.\ P\ p)\ \Rightarrow\ P(\varepsilon p.\ P\ p)$$

[15] Dijkstra's semantics of nondeterministic programs can also be formalized in higher order logic, but not using the simple methods described in this paper (see the end of Section 5.1).

Thus, if there exists a p such that $P\ p$, then $\varepsilon p.\ P\ p$ denotes such a p; if no such p exists, then $\varepsilon p.\ P\ p$ denotes an arbitrary predicate. Hilbert invented ε and showed that it could be consistently added to first order logic. Allowing the use of ε in higher order logic is equivalent to assuming the Axiom of Choice. The weakest predicate satisfying P can be defined using ε:

$$\mathsf{Weakest}\ P \quad = \quad \varepsilon p.\ P\ p \quad \wedge \quad \forall p'.\ P\ p' \ \Rightarrow\ (p{\Leftarrow}p')$$

Dijkstra's two kinds of weakest preconditions can be defined by:

$$\mathsf{Wlp}(c, q) \quad = \quad \mathsf{Weakest}(\lambda p.\ \mathsf{Spec}(p, c, q))$$

$$\mathsf{Wp}(c, q) \quad = \quad \mathsf{Weakest}(\lambda p.\ \mathsf{Total_Spec}(p, c, q))$$

These definitions seems to formalize the intuitive notions described by Dijkstra, but are cumbersome to work with. The theorems shown below are easy consequences of the definitions above, and are much more convenient to use in formal proofs.

$$\vdash\ \mathsf{Wlp}(c, q) \quad = \quad \lambda s.\ \forall s'.\ c(s, s') \ \Rightarrow\ q\ s'$$

$$\vdash\ \mathsf{Wp}(c, q) \quad = \quad \lambda s.\ (\exists s'.\ c(s, s')) \quad \wedge \quad \forall s'.\ c(s, s') \ \Rightarrow\ q\ s'$$

The relationship between Hoare's notation and weakest preconditions is given by:

$$\vdash\ \mathsf{Spec}(p, c, q) \quad = \quad \forall s.\ p\ s \ \Rightarrow\ \mathsf{Wlp}(c, q)\ s$$

$$\vdash\ \mathsf{Total_Spec}(p, c, q) \quad = \quad \forall s.\ p\ s \ \Rightarrow\ \mathsf{Wp}(c, q)\ s$$

The statement of the last two theorems, as well as other results below, can be improved if 'big' versions of the logical operators $\wedge$, $\vee$ $\Rightarrow$ and $\neg$, and constants T and F are introduced which are 'lifted' to predicates. These are defined in the table below, together with the operator $\models$ which tests whether a predicate is always true. These lifted predicates will also be useful in connection with dynamic logic.

Operators on predicates		
$p \bigwedge q$	$=$	$\lambda s.\ p\ s\ \wedge\ q\ s$
$p \bigvee q$	$=$	$\lambda s.\ p\ s\ \vee\ q\ s$
$p \Rrightarrow q$	$=$	$\lambda s.\ p\ s\ \Rightarrow\ q\ s$
$\neg p$	$=$	$\lambda s.\ \neg p\ s$
T	$=$	$\lambda s.\ \mathsf{T}$
F	$=$	$\lambda s.\ \mathsf{F}$
$\models p$	$=$	$\forall s.\ p\ s$

The last two theorems can now be reformulated more elegantly as:

$$\vdash\ \mathsf{Spec}(p, c, q) \quad = \quad \models p \Rrightarrow \mathsf{Wlp}(c, q)$$

$$\vdash\ \mathsf{Total_Spec}(p, c, q) \quad = \quad \models p \Rrightarrow \mathsf{Wp}(c, q)$$

In Dijkstra's book, various properties of weakest preconditions are stated as axioms, for example:

Property 1. $\vdash \forall c.\ \models \mathsf{Wp}(c, \mathsf{F})\ =\ \mathsf{F}$

Property 2. $\vdash \forall q\ r\ c.\ \models (q \Rightarrow r)\ \ \Rightarrow\ \ (\mathsf{Wp}(c,q) \Rightarrow \mathsf{Wp}(c,r))$

Property 3. $\vdash \forall q\ r\ c.\ \models \mathsf{Wp}(c,q) \wedge \mathsf{Wp}(c,r)\ \ =\ \ \mathsf{Wp}(c,\ q{\wedge}r)$

Property 4. $\vdash \forall q\ r\ c.\ \models \mathsf{Wp}(c,q) \vee \mathsf{Wp}(c,r) \Rightarrow \mathsf{Wp}(c,\ q{\vee}r)$

Property 4′. $\vdash \forall q\ r\ c.\ \mathsf{Det}\ c\ \ \Rightarrow\ \models \mathsf{Wp}(c,q) \vee \mathsf{Wp}(c,r)\ =\ \mathsf{Wp}(c,\ q{\vee}r)$

These all follow easily from the definition of Wp given above (Det is the determinacy predicate defined in Section 10). It is also straightforward to derive analogous properties of weakest liberal preconditions:

$\vdash \forall c.\ \models \mathsf{Wlp}(c, \mathsf{F})\ =\ \lambda s.\ \neg\exists s'.\ c(s,s')$

$\vdash \forall q\ r\ c.\ \models (q \Rightarrow r)\ \ \Rightarrow\ \ (\mathsf{Wlp}(c,q) \Rightarrow \mathsf{Wlp}(c,r))$

$\vdash \forall q\ r\ c.\ \models \mathsf{Wlp}(c,q) \wedge \mathsf{Wlp}(c,r)\ =\ \mathsf{Wlp}(c,\ q{\wedge}r)$

$\vdash \forall q\ r\ c.\ \models \mathsf{Wlp}(c,q) \vee \mathsf{Wlp}(c,r) \Rightarrow \mathsf{Wlp}(c,\ q{\vee}r)$

$\vdash \forall q\ r\ c.\ \mathsf{Det}\ c\ \ \Rightarrow\ \models \mathsf{Wlp}(c,q) \vee \mathsf{Wlp}(c,r)\ =\ \mathsf{Wlp}(c,\ q{\vee}r)$

Many of the properties of programming constructs given in Dijkstra's book [6] are straightforward to verify for the constructs of our little language. For example:

$\vdash\ \mathsf{Wp}([\![\mathbf{skip}]\!],\ q) = q$

$\vdash\ \mathsf{Wlp}([\![\mathbf{skip}]\!],\ q) = q$

$\vdash\ \mathsf{Wp}([\![\mathcal{V}\ \texttt{:=}\ \mathcal{E}]\!],\ q) = \lambda s.\ q(\mathsf{Bnd}\ ([\![\mathcal{E}]\!]s)\ \text{‘}\mathcal{V}\text{‘}\ s)$

$\vdash\ \mathsf{Wlp}([\![\mathcal{V}\ \texttt{:=}\ \mathcal{E}]\!],\ q) = \lambda s.\ q(\mathsf{Bnd}\ ([\![\mathcal{E}]\!]s)\ \text{‘}\mathcal{V}\text{‘}\ s)$

$\vdash\ \mathsf{Wp}([\![\mathbf{if}\ \mathcal{B}\ \mathbf{then}\ \mathcal{C}_1\ \mathbf{else}\ \mathcal{C}_2]\!],\ q) = \lambda s.\ ([\![\mathcal{B}]\!]s \rightarrow \mathsf{Wp}([\![\mathcal{C}_1]\!],s) \mid \mathsf{Wp}([\![\mathcal{C}_2]\!],s))$

$\vdash\ \mathsf{Wlp}([\![\mathbf{if}\ \mathcal{B}\ \mathbf{then}\ \mathcal{C}_1\ \mathbf{else}\ \mathcal{C}_2]\!],\ q) = \lambda s.\ ([\![\mathcal{B}]\!]s \rightarrow \mathsf{Wlp}([\![\mathcal{C}_1]\!],s) \mid \mathsf{Wlp}([\![\mathcal{C}_2]\!],s))$

The inadequacy of the relational model reveals itself when we try to derive Dijkstra's Wp-law for sequences. This law is:

$$\mathsf{Wp}([\![\mathcal{C}_1\,;\ \mathcal{C}_2]\!],\ q) = \mathsf{Wp}([\![\mathcal{C}_1]\!],\ \mathsf{Wp}([\![\mathcal{C}_2]\!],\ q))$$

which is not true with our semantics. For example, taking:

$$\begin{array}{ll}
\mathsf{s}_1 & = \lambda x.\ 0 \\
\mathsf{s}_2 & = \lambda x.\ 1 \\
\mathsf{c}_1(s_1,s_2) & = (s_1 = \mathsf{s}_1)\ \vee\ (s_2 = \mathsf{s}_2) \\
\mathsf{c}_2(s_1,s_2) & = (s_1 = \mathsf{s}_1)\ \wedge\ (s_2 = \mathsf{s}_2)
\end{array}$$

results in:

$$\mathsf{Wp}(\mathsf{Seq}(\mathsf{c}_1,\mathsf{c}_2),\mathsf{T}) = \mathsf{T}$$

but

$$\mathsf{Wp}(\mathsf{c}_1,\ \mathsf{Wp}(\mathsf{c}_2,\mathsf{T}))\ = \mathsf{F}$$

The best that can be proved using the relational semantics is the following:

$$\vdash\ \mathsf{Det}\ [\![\mathcal{C}_1]\!]\quad\Rightarrow\quad \mathsf{Wp}([\![\mathcal{C}_1\,;\ \mathcal{C}_2]\!],\ q) = \mathsf{Wp}([\![\mathcal{C}_1]\!],\ \mathsf{Wp}([\![\mathcal{C}_2]\!],\ q))$$

As discussed in Section 5.1, the problem lies in the definition of Halts. For partial correctness there is no problem; the following sequencing law for weakest liberal preconditions can be proved from the relational semantics.

$$\vdash\ \mathsf{Wlp}([\![\mathcal{C}_1\,;\ \mathcal{C}_2]\!],\ q) = \mathsf{Wlp}([\![\mathcal{C}_1]\!],\ \mathsf{Wlp}([\![\mathcal{C}_2]\!],\ q))$$

With relational semantics, the Wp-law for **while**-commands also requires a determinacy assumption:

$$\vdash\ \mathsf{Det}\ c\quad\Rightarrow\quad \mathsf{Wp}([\![\mathbf{while}\ \mathcal{B}\ \mathbf{do}\ \mathcal{C}]\!],\ q)\ s\ =\ \exists n.\ \mathsf{Iter_Wp}\ n\ [\![\mathcal{B}]\!]\ [\![\mathcal{C}]\!]\ q\ s$$

where

$$\begin{array}{lcl}\mathsf{Iter_Wp}\ 0\ b\ c\ q & = & \neg b \wedge p\\ \mathsf{Iter_Wp}\ (n{+}1)\ b\ c\ q & = & b \wedge \mathsf{Wp}(c,\ \mathsf{Iter_Wp}\ n\ b\ c\ p)\end{array}$$

However, the Wlp-law for **while**-commands does not require a determinacy assumption:

$$\vdash\ \mathsf{Wlp}([\![\mathbf{while}\ \mathcal{B}\ \mathbf{do}\ \mathcal{C}]\!],\ q)\ s\ =\ \forall n.\ \mathsf{Iter_Wlp}\ n\ [\![\mathcal{B}]\!]\ [\![\mathcal{C}]\!]\ q\ s$$

where

$$\begin{array}{lcl}\mathsf{Iter_Wlp}\ 0\ b\ c\ q & = & \neg b\ \Rightarrow\ p\\ \mathsf{Iter_Wlp}\ (n{+}1)\ b\ c\ q & = & b\ \Rightarrow\ \mathsf{Wlp}(c,\ \mathsf{Iter_Wlp}\ n\ b\ c\ p)\end{array}$$

The Wlp-law for **while**-commands given above was not the first one I thought of. Initially I tried the same tactic that was used to prove the Wp-law for **while**-commands on the goal obtained from it by deleting the determinacy assumption and replacing Wp by Wlp. It soon became clear that this would not work, and after some 'proof hacking' with HOL I came up with the theorem above. A possible danger of powerful proof assistants is that they will encourage the generation of theorems without much understanding of their significance. I tend to use HOL for simple mathematics rather like I use a pocket calculator for arithmetic. I have already forgotten how to do some arithmetic operations by hand; I hope HOL will not cause me to forget how to do mathematical proofs manually!

11.3 Dynamic logic

Dynamic logic is a programming logic which emphasizes an analogy between Hoare logic and modal logic; it was invented by V.R. Pratt based on an idea of R.C. Moore [32, 9]. In dynamic logic, states of computation are thought of as *possible worlds*, and if a command $\mathcal{C}$ transforms an initial state s to a final state s' then s' is thought of as *accessible* from s (the preceding phrases in italics are standard concepts from modal logic).

Modal logic is characterized by having formulae $\Box q$ and $\Diamond q$ with the following interpretations.

- $\Box q$ is true in s if q is true in all states accessible from s.
- $\Diamond q$ is true in s if $\neg\Box\neg q$ is true in s.

Instead of a single $\Box$ and $\Diamond$, dynamic logic has operators $[\mathcal{C}]$ and $<\mathcal{C}>$ for each command $\mathcal{C}$. These can be defined on the relation c denotated by $\mathcal{C}$ as follows:

$$[c]\,q \quad = \quad \lambda s.\ \forall s'.\ c(s,s') \ \Rightarrow\ q\ s'$$

$$<c>q \quad = \quad \neg([c]\,(\neg q))$$

where $\neg$ is negation lifted to predicates (see preceding section).

A typical theorem of dynamic logic is:

$$\vdash\ \forall c\ q.\ \mathsf{Det}\ c \ \ \Rightarrow\ \models <c>q \Rrightarrow\ [c]\,q$$

This is a version of the modal logic principle that says that if the accessibility relation is functional then $\Diamond q\ \Rightarrow\ \Box q$ [9].

From the definitions of $[c]\,q$ and $<c>q$ it can be easily deduced that:

$$\vdash\ (\models [c]\,q)\ =\ \mathsf{Spec}(\mathsf{T}, c, q)$$

$$\vdash\ \mathsf{Det}\ c \ \ \Rightarrow\ \ ((\models <c>q)\ =\ \mathsf{Total_Spec}(\mathsf{T}, c, q))$$

$$\vdash\ \mathsf{Spec}(p, c, q)\ =\ (\models p \Rrightarrow\ [c]\,q)$$

$$\vdash\ \mathsf{Det}\ c \ \ \Rightarrow\ \ (\mathsf{Total_Spec}(p, c, q)\ =\ (\models p \Rrightarrow\ <c>q))$$

Where $\models$, $\Rrightarrow$ and T were defined in the preceding section. Using these relationships, theorems of dynamic logic can be converted to theorems of Hoare logic (and vice versa).

Dynamic logic is closely related to weakest preconditions as follows:

$$\vdash\ \mathsf{Wlp}(c, q)\ =\ [c]\,q$$

$$\vdash\ \mathsf{Det}\ c \ \ \Rightarrow\ \ (\mathsf{Wp}(c, q)\ =\ <c>q)$$

These theorems can be used to translate results from one system to the other.

12 Conclusions and future work

The examples in the previous section show that it is straightforward to define the semantic content of diverse programming logics directly in higher order logic. In earlier sections it is shown how, with a modest amount of parsing and pretty printing, this semantic representation can be made syntactically palatable. If a general purpose system like HOL is used, the choice of specification constructs can be optimized to the problem at hand and to the tastes of the specifier; a particular choice need not be hard-wired into the verifier. For example, Hoare-style and VDM-style correctness specifications can be freely mixed. A significant benefit of working in a single logical system is that only a single set of theorem proving tools is needed. Typical software verifications require some general mathematical reasoning, as well as specialized manipulations in a programming logic. If everything is embedded in a single logic, then there is no need to interface a special purpose program verifier to a separate theorem prover for handling verification conditions. This benefit is even greater if both software and hardware are being simultaneously reasoned about, because hardware verification tools can also be embedded in systems like HOL [12].

Although the methods presented here seem to work smoothly, the examples done so far are really too trivial to permit firm conclusions to be drawn. The next step in this research is to try to develop a *practical* program verifier on top of the HOL system. We plan to extend the methods of this paper to a programming language containing at least procedures, functions, arrays and some input/output. Various possibilities are under consideration, ranging from adopting an existing language like Tempura, Occam or Vista, to designing our own verification-oriented language. It is intended that whatever language we choose will be supported by a verified compiler generating code for a verified processor. Preliminary work on this has already started. Eventually it is planned to verify a reasonably non-trivial program using our tools. It is expected that this case study will be some kind of simple real-time system. Our goal is to show the *possibility* of totally verified systems, and to give a preliminary idea of their practicability.

One unsatisfactory aspect of the verifier described here is that the parser and pretty-printer were implemented by descending from ML into Lisp (the HOL system is implemented in Lisp). This could be avoided if ML were augmented with a general purpose parser and pretty-printer generator similar to the one in Mosses' semantics implementation system SIS [26]. Providing these tools would be quite a lot of work, but fortunately some progress in this direction has already been made. For example:

(i) Huet's group at INRIA have interfaced CAML (a version of ML that extends the ML used by HOL) to the Unix YACC parser generator.

(ii) It is planned that a parser generator (from Edinburgh) will be distributed with *Standard ML of New Jersey*, a high performance implementation of Standard ML from AT&T Bell Laboratories.

(iii) The Esprit project *GIPE* (*G*enerating *I*nteractive *P*rogramming *E*nvironments) is producing a powerful general purpose interface for manipulating formal languages. This supports a user-specifiable parser and pretty-printer which is closely integrated with a mouse-driven syntax-directed editor. For example, the pretty-printer adapts its line breaks to the width of the window in which it is being used.

These three projects suggest that tools will soon be available to enable syntactic interfaces to logical systems to be smoothly implemented without the low-level hacking I had to use.

13 Acknowledgements

The style of interactive proof supported by the HOL system was invented by Robin Milner and extended by Larry Paulson. The idea of representing verification condition generation by a tactic was developed jointly with Tom Melham; he also spotted errors in an earlier version of this paper. Job Zwiers, of the Philips Research Laboratories in Eindhoven, explained to me the connection between determinacy, relational semantics, dynamic logic and Dijkstra's weakest preconditions. Avra Cohn provided techical advice, as well as general support and encouragement. Finally, the past and present members of the Cambridge hardware verification group generated the exciting intellectual environment in which the research reported here was conducted.

References

[1] Andrews, P.B., *An Introduction to Mathematical Logic and Type Theory*, Academic Press, 1986.

[2] Boyer, R.S. and Moore, J S., *A Computational Logic*, Academic Press, 1979.

[3] Boyer, R.S., and Moore, J S., 'Metafunctions: proving them correct and using them efficiently as new proof procedures' in Boyer, R.S. and Moore, J S. (eds), *The Correctness Problem in Computer Science*, Academic Press, New York, 1981.

[4] Clarke, E.M. Jr., 'The characterization problem for Hoare logics', in Hoare, C.A.R. and Shepherdson, J.C. (eds), *Mathematical Logic and Programming Languages*, Prentice Hall, 1985.

[5] A. Church, 'A Formulation of the Simple Theory of Types', Journal of Symbolic Logic **5**, 1940.

[6] Dijkstra, E.W., *A Discipline of Programming*, Prentice-Hall, 1976.

[7] Floyd, R.W., 'Assigning meanings to programs', in Schwartz, J.T. (ed.), *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics* **19** (American Mathematical Society), Providence, pp. 19-32, 1967.

[8] Good, D.I., 'Mechanical proofs about computer programs', in Hoare, C.A.R. and Shepherdson, J.C. (eds), *Mathematical Logic and Programming Languages*, Prentice Hall, 1985.

[9] Goldblatt, R., *Logics of Time and Computation*, CSLI Lecture Notes **7**, CSLI/Stanford, Ventura Hall, Stanford, CA 94305, USA, 1987.

[10] Gordon, M.J.C., 'Representing a logic in the LCF metalanguage', in Néel, D. (ed.), *Tools and Notions for Program Construction*, Cambridge University Press, 1982.

[11] Gordon, M.J.C., Milner, A.J.R.G. and Wadsworth, C.P., *Edinburgh LCF: a mechanized logic of computation*, Springer Lecture Notes in Computer Science **78**, Springer-Verlag, 1979.

[12] M. Gordon, 'Why Higher-order Logic is a Good Formalism for Specifying and Verifying Hardware', in G. Milne and P. A. Subrahmanyam (eds), *Formal Aspects of VLSI Design*, North-Holland, 1986.

[13] Gordon, M.J.C., 'HOL: A Proof Generating System for Higher-Order Logic', University of Cambridge, Computer Laboratory, Tech. Report No. 103, 1987; Revised version in G. Birtwistle and P.A. Subrahmanyam (eds), *VLSI Specification, Verification and Synthesis*, Kluwer, 1987.

[14] Gordon, M.J.C., *Programming Language Theory and its Implementation*, Prentice-Hall, 1988.

[15] Gries, D., *The Science of Programming*, Springer-Verlag, 1981.

[16] Hoare, C.A.R., 'An axiomatic basis for computer programming', *Communications of the ACM* **12**, pp. 576-583, October 1969.

[17] Igarashi, S., London, R.L., Luckham, D.C., 'Automatic program verification I: logical basis and its implementation', Acta Informatica **4**, 1975, pp. 145-182.

[18] INMOS Limited, 'Occam Programming Language', Prentice-Hall.

[19] Jones, C.B., 'Systematic Program Development' in Gehani, N. & McGettrick, A.D. (eds), *Software Specification Techniques*, Addison-Wesley, 1986.

[20] Joyce, J.J., Forthcoming Ph.D. thesis, University of Cambridge Computer Laboratory, expected 1989.

[21] Ligler, G.T., 'A mathematical approach to language design', in *Proceedings of the Second ACM Symposium on Principles of Programming Languages*, pp. 41-53.

[22] Loeckx, J. and Sieber, K., *The Foundations of Program Verification*, John Wiley & Sons Ltd. and B.G. Teubner, Stuttgart, 1984.

[23] London, R.L., et al. 'Proof rules for the programming language Euclid', *Acta Informatica* **10**, No. 1, 1978.

[24] Fourman, M.P., 'The Logic of Topoi', in Barwise, J. (ed.), *Handbook of Mathematical Logic*, North-Holland, 1977.

[25] Melham. T.F., 'Automating Recursive Type Definitions in Higher Order Logic', in Birtwistle, G. and P.A. Subrahmanyam (eds), Proceedings of the *1988 Banff Conference on Hardware Verification* (exact book title unknown), Springer-Verlag, 1988.

[26] Mosses, P.D., 'Compiler Generation using Denotational Semantics', in *Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science **45**, Springer-Verlag, 1976.

[27] Milner, A.R.J.G., 'A Theory of Type Polymorphism in Programming', *Journal of Computer and System Sciences* **17**, 1978.

[28] Paulson, L.C., 'A higher-order implementation of rewriting', *Science of Computer Programming* **3**, pp 143-170, 1985.

[29] Paulson, L.C., 'Natural deduction as higher-order resolution', *Journal of Logic Programming* **3**, pp 237-258, 1986.

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

[31] Plotkin, G.D., 'Dijkstra's Predicate Transformers and Smyth's Powerdomains', in Bjørner, D. (ed.), *Abstract Software Specifications*, Lecture Notes in Computer Science **86**, Springer-Verlag, 1986.

[32] Pratt, V.R., 'Semantical Considerations on Floyd-Hoare Logic', *Proceedings of the 17th IEEE Symposium on Foundations of Computer Science*, 1976.

[33] Hayes, I. (ed.), *Specification Case Studies*, Prentice-Hall.

Index

$\{\mathcal{P}\}\ \mathcal{C}\ \{\mathcal{Q}\}$, 5
$[\mathcal{P}]\ \mathcal{C}\ [\mathcal{Q}]$, 5
$\Box$, 49
[$\mathcal{C}$], 49
$\Diamond$, 49
<$\mathcal{C}$>, 49
T, 10
T, 46
F, 10
F, 46
$\neg$, 10
$\neg$, 46
$\wedge$, 10
$\bigwedge$, 46
$\vee$, 10
$\bigvee$, 46
$\Rightarrow$, 10
$\Rrightarrow$, 46
$\Leftarrow$, 45
$\models$, 46
$\|$, 16
$[\![\cdots]\!]$, 14
$(\cdots \rightarrow \cdots \mid \cdots)$, 10

accessibility relation, 49
Aczel, P., 43
annotation, 33
Assign, 15
ASSIGN_AX, 23
ASSIGN_TAC, 33
assignment axiom, 6
    as an ML function, 23
auxiliary variable, 14
Axiom of Choice, 46

Bnd, 15
Boyer-Moore theorem prover, 22

CAML, 50
Choose, 16
Church's type system, 11
closed term, 13
command, 4
    semantics in HOL, 13, 15

deep structure, 18
definitions in HOL, 13
Det, 39
determinacy, 16, 39
    and sequencing, 48
Dijkstra, E.W., 45, 46
dynamic logic, 49
    in HOL, 49

expand, 31

forward proof, 9
Fourman, M.P., 13

ghost variable, 14
GIPE, 51
goal oriented proof, 9
goal, 31
goals, 9
    achievement of, 29
    solution by a tactic, 30

Halts, 38
higher order functions, 10
higher order logic, 10
higher order unification, 19
Hilbert's ε-operator, 45
Hoare logic, 5
    axioms of, 6
    example proof, 8
    in HOL, 17
    rules of inference of, 6
HOL, 22
    and LCF, 11
    user interface, 22
Huet, G, 50

If, 15
IF_TAC, 33
if-rule, 7
    as an ML function, 25
    derivation of, 20
IF_RULE, 25
invariant, 7, 33
Isabelle, 19
Iter, 15
Iter_Wlp, 48
Iter_Wp, 48

Jones, C., 43

λ-expression, 11

Lisp, 22
logical variable, 14

Melham, T., 13, 39, 51
meta notation in HOL, 22
Milner, R., 12
ML, 22
Moore, R.C., 49
Mosses, P, 50

nondeterministic commands, 16

Occam, 50
ORELSE, 31

partial correctness, 5
  semantics in HOL, 17
Pascal, 7
Paulson, L.C., 30
possible world, 49
postcondition, 5
postcondition weakening, 7
  as an ML function, 24
  derivation of, 19
POST_WEAK_RULE, 24
Pratt, V.R., 49
precondition, 5
precondition strengthening, 7
  as an ML function, 24
  derivation of, 19
predicate calculus notation, 10
PRE_STRENGTH_RULE, 24

REPEAT, 31
Russell's paradox, 11

Seq, 11, 15
SEQ_RULE, 23
SEQ_TAC, 33
SEQL_RULE, 24
sequencing rule, 7
  as an ML function, 23
  derivation of, 20
sequential boolean operators, 7
Skip, 15
SKIP_TAC, 32
skip-axiom, 6
  derivation of, 18
Spec, 11
Standard ML, 50
subgoal, 9
surface structure, 18

TAC_PROOF, 30
tactic, 9, 26, 29
tactical, 29, 31
Tempura, 50
termination, 5, 38
THEN, 31
THENL, 31
theories in HOL, 13
total correctness, 5, 38
  derived rules in HOL, 40
  tactics for, 40
  verification conditions for, 41
Total_Spec, 39
Transitive, 45
type inference, 12
type operators, 12
type theory, 11
types in HOL, 11
  atomic, 11
  compound, 11

validations in HOL, 29
variant, 39, 41
VDM, 43
  in HOL, 43
verification condition, 9
  and weakest liberal precondition, 32
  via tactics, 31
Vista, 50

weakest liberal precondition, 32, 45
weakest precondition, 45
Weakest, 46
well-typed, 12
Well_Founded, 45
While, 15
WHILE_TAC, 33
while-rule, 7
  as an ML function, 25
  derivation of, 20
WHILE_RULE, 25
Wlp, 32, 45
Wp, 45

Zwiers, J., 51