Title: *Specification and Verification I*

Lecturer: **Mike Gordon**

Class: **Computer Science Tripos, Part II**

Duration: **Twelve lectures**

December 11, 2009

Specification and Verification I

Mike Gordon

Overview

These lecture notes are for the course entitled *Specification and Verification I*. Some of the material is derived from previously published sources.[1]

Chapters 1–4 introduce classical ideas of specification and proof of programs due to Floyd and Hoare. Chapter 5 is an introduction to program refinement using an approach due to Paul Curzon. Chapter 6 presents higher order logic and Chapter 7 explains how Floyd-Hoare logic can be embedded in higher order logic.

The course presents classical ideas on the specification and verification of software. Although much of the material is old – see the dates on some of the cited references – it is still a key foundation for current research.[2]

This course is a prerequisite for the Part II course entitled *Specification and Verification II*, which makes extensive use of higher order logic (see Chapter 6) for specifying and verifying hardware.

Learning Guide

These notes contain all the material that will be covered in the course. It should thus not be necessary to consult any textbooks etc.

The copies of transparencies give the contents of the lectures. However note that I sometimes end up going faster or slower than expected so, for example, material shown in Lecture n might actually get covered in Lecture $n{+}1$ or Lecture $n{-}1$.

The examination questions will be based on material in the lectures. Thus if I end up not covering some topic in the lectures, then I would not expect to set an examination question on it.

This course has been fairly stable for several years, so past exam questions are a reasonable guide to the sort of thing I will set this year.

Acknowledgement

Paul Curzon made many improvements, extensions and corrections to these notes.

[1] M.J.C. Gordon, *Programming Language Theory and its Implementation*, Prentice-Hall International Series in Computer Science (edited by Professor C.A.R Hoare), 1988 (out of print); M.J.C. Gordon, *Mechanizing Programming Logics in Higher Order Logic*, in *Current Trends in Hardware Verification and Automated Theorem Proving*, edited by G. Birtwistle and P.A. Subrahmanyam, Springer-Verlag, 1989.

[2] See `www.dcs.qmul.ac.uk/~ohearn/localreasoning.html` for a current research thread.

Contents

Chapter 1

Program Specification

A simple programming language containing assignments, conditionals, blocks, `WHILE`-commands and `FOR`-commands is introduced. This language is then used to illustrate Hoare's notation for specifying the partial correctness of programs. Hoare's notation uses predicate calculus to express conditions on the values of program variables. A fragment of predicate calculus is introduced and illustrated with examples.

1.1 Introduction

In order to prove mathematically the correctness of a program one must first specify what it means for it to be correct. In this chapter a notation for specifying the desired behaviour of *imperative* programs is described. This notation is due to C.A.R. Hoare.

Executing an imperative program has the effect of changing the *state*, i.e. the values of program variables[1]. To use such a program, one first establishes an initial state by setting the values of some variables to values of interest. One then executes the program. This transforms the initial state into a final one. One then inspects (using print commands etc.) the values of variables in the final state to get the desired results. For example, to compute the result of dividing `y` into `x` one might load `x` and `y` into program variables `X` and `Y`, respectively. One might then execute a suitable program (see Example 7 in Section 1.4) to transform the initial state into a final state in which the variables `Q` and `R` hold the quotient and remainder, respectively.

The programming language used in these notes is described in the next section.

1.2 A little programming language

Programs are built out of *commands* like assignments, conditionals etc. The terms 'program' and 'command' are really synonymous; the former will only be used for commands representing complete algorithms. Here the term 'statement' is used

[1] For languages more complex than those described here, the state may consist of other things besides the values of variables [16].

for conditions on program variables that occur in correctness specifications (see Section 1.3). There is a potential for confusion here because some writers use this word for commands (as in 'for-statement' [21]).

We now describe the *syntax* (i.e. form) and *semantics* (i.e. meaning) of the various commands in our little programming language. The following conventions are used:

1. The symbols V, V_1, ... , V_n stand for arbitrary variables. Examples of particular variables are `X`, `R`, `Q` etc.

2. The symbols E, E_1, ... , E_n stand for arbitrary expressions (or terms). These are things like $\mathtt{X} + \mathtt{1}$, $\sqrt{2}$ etc. which denote values (usually numbers).

3. The symbols S, S_1, ... , S_n stand for arbitrary statements. These are conditions like $\mathtt{X} < \mathtt{Y}$, $\mathtt{X}^2 = \mathtt{1}$ etc. which are either true or false.

4. The symbols C, C_1, ... , C_n stand for arbitrary commands of our programming language; these are described in the rest of this section.

Terms and statements are described in more detail in Section 1.5.

1.2.1 Assignments

Syntax: V `:=` E

Semantics: The state is changed by assigning the value of the term E to the variable V.

Example: `X:=X+1`

This adds one to the value of the variable `X`.

1.2.2 Array assignments

Syntax: $V(E_1)$`:=`E_2

Semantics: The state is changed by assigning the value of the term E_2 to the E_1th component of the array variable V.

Example: `A(X+1) := A(X)+2`

If the the value of `X` is n then the value stored in the n+1th component of `A` becomes the value of the nth component of `A` plus `2`.

1.2.3 Sequences

Syntax: C_1; $\cdots$;C_n

Semantics: The commands $C_1, \cdots, C_n$ are executed in that order.

Example: `R:=X; X:=Y; Y:=R`

> The values of `X` and `Y` are swapped using `R` as a temporary variable. This command has the *side effect* of changing the value of the variable `R` to the old value of the variable `X`.

1.2.4 Blocks

Syntax: `BEGIN VAR` V_1; $\cdots$ `VAR` V_n; C `END`

Semantics: The command C is executed, and then the values of $V_1, \cdots, V_n$ are restored to the values they had before the block was entered. The initial values of $V_1, \cdots, V_n$ inside the block are unspecified.

Example: `BEGIN VAR R; R:=X; X:=Y; Y:=R END`

> The values of `X` and `Y` are swapped using `R` as a temporary variable. This command does *not* have a side effect on the variable `R`.

1.2.5 One-armed conditionals

Syntax: `IF` S `THEN` C

Semantics: If the statement S is true in the current state, then C is executed. If S is false, then nothing is done.

Example: `IF ¬(X=0) THEN R:= Y DIV X`

> If the value `X` is not zero, then `R` is assigned the result of dividing the value of `Y` by the value of `X`.

1.2.6 Two-armed conditionals

Syntax: `IF` S `THEN` C_1 `ELSE` C_2

Semantics: If the statement S is true in the current state, then C_1 is executed. If S is false, then C_2 is executed.

Example: `IF X<Y THEN MAX:=Y ELSE MAX:=X`

> The value of the variable `MAX` it set to the maximum of the values of `X` and `Y`.

1.2.7 `WHILE`-commands

Syntax: `WHILE` S `DO` C

Semantics: If the statement S is true in the current state, then C is executed and the `WHILE`-command is then repeated. If S is false, then nothing is done. Thus C is repeatedly executed until the value of S becomes false. If S never becomes false, then the execution of the command never terminates.

Example: `WHILE ¬(X=0) DO X:= X-2`

> If the value of `X` is non-zero, then its value is decreased by `2` and then the process is repeated. This `WHILE`-command will terminate (with `X` having value `0`) if the value of `X` is an even non-negative number. In all other states it will not terminate.

1.2.8 `FOR`-commands

Syntax: `FOR` V`:=`E_1 `UNTIL` E_2 `DO` C

Semantics: If the values of terms E_1 and E_2 are positive numbers e_1 and e_2 respectively, and if $e_1 \leq e_2$, then C is executed $(e_2-e_1)+1$ times with the variable V taking on the sequence of values e_1, e_1+1, ... , e_2 in succession. For any other values, the `FOR`-command has no effect. A more precise description of this semantics is given in Section 2.1.11.

Example: `FOR N:=1 UNTIL M DO X:=X+N`

> If the value of the variable `M` is m and $m \geq 1$, then the command `X:=X+N` is repeatedly executed with `N` taking the sequence of values `1`, ... , m. If $m < 1$ then the `FOR`-command does nothing.

1.2.9 Summary of syntax

The syntax of our little language can be summarized with the following specification in BNF notation[2]

```
<command>
   ::=  <variable>:=<term>
     |  <variable>(<term>):=<term>
     |  <command>; ... ;<command>
     |  BEGIN VAR <variable>; ... VAR <variable>; <command> END
     |  IF <statement> THEN <command>
     |  IF <statement> THEN <command> ELSE <command>
     |  WHILE <statement> DO <command>
     |  FOR <variable>:=<term> UNTIL <term> DO <command>
```

[2]BNF stands for Backus-Naur form; it is a well-known notation for specifying syntax.

Note that:

- *Variables*, *terms* and *statements* are as described in Section 1.5.
- Only declarations of the form '`VAR` <*variable*>' are needed. The types of variables need not be declared (unlike in Pascal).
- Sequences C_1`;` ... C_n are valid commands; they are equivalent to `BEGIN` C_1`;` ... C_n `END` (i.e. blocks without any local variables).
- The BNF syntax is ambiguous: it does not specify, for example, whether `IF` S_1 `THEN IF` S_2 `THEN` C_1 `ELSE` C_2 means

  `IF` S_1 `THEN (IF` S_2 `THEN` C_1 `ELSE` C_2`)`

  or

  `IF` S_1 `THEN (IF` S_2 `THEN` C_1`) ELSE` C_2

  We will clarify, whenever necessary, using brackets.

1.3 Hoare's notation

In a seminal paper [20] C.A.R. Hoare introduced the following notation for specifying what a program does[3]:

$$\{P\}\ C\ \{Q\}$$

where:

- C is a program from the programming language whose programs are being specified (the language in Section 1.2 in our case).
- P and Q are conditions on the program variables used in C.

Conditions on program variables will be written using standard mathematical notations together with *logical operators* like $\wedge$ ('and'), $\vee$ ('or'), $\neg$ ('not') and $\Rightarrow$ ('implies'). These are described further in Section 1.5.

We say $\{P\}\ C\ \{Q\}$ is true, if whenever C is executed in a state satisfying P and if the execution of C terminates, then the state in which C's execution terminates satisfies Q.

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

[3] Actually, Hoare's original notation was $P\ \{C\}\ Q$ not $\{P\}\ C\ \{Q\}$, but the latter form is now more widely used.

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

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

A stronger kind of specification is a *total correctness specification*. There is no standard notation for such specifications. We shall use $[P]\ C\ [Q]$.

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

(i) Whenever C is executed in a state satisfying P, then the execution of C terminates.

(ii) After termination Q holds.

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

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

Total correctness is what we are ultimately interested in, but it is usually easier to prove it by establishing partial correctness and termination separately.

Termination is often straightforward to establish, but there are some well-known examples where it is not. For example[4], no one knows whether the program below terminates for all values of `X`:

```
WHILE X>1 DO
  IF ODD(X) THEN X := (3×X)+1 ELSE X := X DIV 2
```

(The expression `X DIV 2` evaluates to the result of rounding down `X/2` to a whole number.)

Exercise 1

Write a specification which is true if and only if the program above terminates. □

1.4 Some examples

The examples below illustrate various aspects of partial correctness specification.

In Examples 5, 6 and 7 below, `T` (for 'true') is the condition that is always true. In Examples 3, 4 and 7, $\wedge$ is the logical operator 'and', i.e. if P_1 and P_2 are conditions, then $P_1 \wedge P_2$ is the condition that is true whenever both P_1 and P_2 hold.

1. $\{\mathtt{X} = \mathtt{1}\}$ `Y:=X` $\{\mathtt{Y} = \mathtt{1}\}$

[4] This example is taken from Exercise 2 on page 17 of Reynolds's book [38].

This says that if the command `Y:=X` is executed in a state satisfying the condition $\texttt{X} = \texttt{1}$ (i.e. a state in which the value of `X` is 1), then, if the execution terminates (which it does), then the condition $\texttt{Y} = \texttt{1}$ will hold. Clearly this specification is true.

2. $\{\texttt{X} = \texttt{1}\}$ `Y:=X` $\{\texttt{Y} = \texttt{2}\}$

This says that if the execution of `Y:=X` terminates when started in a state satisfying $\texttt{X} = \texttt{1}$, then $\texttt{Y} = \texttt{2}$ will hold. This is clearly false.

3. $\{\texttt{X=x} \wedge \texttt{Y=y}\}$ `BEGIN R:=X; X:=Y; Y:=R END` $\{\texttt{X=y} \wedge \texttt{Y=x}\}$

This says that if the execution of `BEGIN R:=X; X:=Y; Y:=R END` terminates (which it does), then the values of `X` and `Y` are exchanged. The variables `x` and `y`, which don't occur in the command and are used to name the initial values of program variables `X` and `Y`, are called *auxiliary* variables (or *ghost* variables).

4. $\{\texttt{X=x} \wedge \texttt{Y=y}\}$ `BEGIN X:=Y; Y:=X END` $\{\texttt{X=y} \wedge \texttt{Y=x}\}$

This says that `BEGIN X:=Y; Y:=X END` exchanges the values of `X` and `Y`. This is not true.

5. $\{\texttt{T}\}\ C\ \{Q\}$

This says that whenever C halts, Q holds.

6. $\{P\}\ C\ \{\texttt{T}\}$

This specification is true for every condition P and every command C (because `T` is always true).

7. $\{\texttt{T}\}$

$$\left.\begin{array}{l}\texttt{BEGIN}\\ \quad\texttt{R:=X;}\\ \quad\texttt{Q:=0;}\\ \quad\texttt{WHILE Y}\leq\texttt{R DO}\\ \qquad\texttt{BEGIN R:=R-Y; Q:=Q+1 END}\\ \texttt{END}\end{array}\right\} C$$

$\{\texttt{R} < \texttt{Y} \wedge\ \texttt{X} = \texttt{R} + (\texttt{Y} \times \texttt{Q})\}$

This is $\{\texttt{T}\}\ C\ \{\texttt{R} < \texttt{Y}\ \wedge\ \texttt{X} = \texttt{R} + (\texttt{Y} \times \texttt{Q})\}$ where C is the command indicated by the braces above. The specification is true if whenever the execution of C halts, then `Q` is quotient and `R` is the remainder resulting from dividing `Y` into `X`. It is true (even if `X` is initially negative!).

In this example a program variable `Q` is used. This should not be confused with the Q used in 5 above. The program variable `Q` (notice the font) ranges over numbers, whereas the postcondition Q (notice the font) ranges over statements. In general, we use `typewriter font` for particular program variables and *italic font* for variables ranging over statements. Although this subtle use of fonts might appear confusing at first, once you get the hang of things the difference between the two kinds of 'Q' will be clear (indeed you should be able to disambiguate things from context without even having to look at the font).

Exercise 2
Let C be as in Example 7 above. Find a condition P such that:

$$[P]\ C\ [\mathtt{R} < \mathtt{Y} \wedge \mathtt{X} = \mathtt{R} + (\mathtt{Y} \times \mathtt{Q})]$$

is true. □

Exercise 3
When is $[\mathtt{T}]\ C\ [\mathtt{T}]$ true? □

Exercise 4
Write a partial correctness specification which is true if and only if the command C has the effect of multiplying the values of X and Y and storing the result in X. □

Exercise 5
Write a specification which is true if the execution of C always halts when execution is started in a state satisfying P. □

1.5 Terms and statements

The notation used here for expressing pre- and postconditions is based on first-order logic. This will only be briefly reviewed here as readers are assumed to be familiar with it. [5]

The following are examples of *atomic statements*.

$$\mathtt{T}, \qquad \mathtt{F}, \qquad \mathtt{X} = \mathtt{1}, \qquad \mathtt{R} < \mathtt{Y}, \qquad \mathtt{X} = \mathtt{R+(Y{\times}Q)}$$

Statements are either true or false. The statement T is always true and the statement F is always false. The statement $\mathtt{X} = \mathtt{1}$ is true if the value of X is equal to 1. The statement $\mathtt{R} < \mathtt{Y}$ is true if the value of R is less than the value of Y. The statement $\mathtt{X} = \mathtt{R+(Y{\times}Q)}$ is true if the value of X is equal to the sum of the value of R with the product of Y and Q.

Statements are built out of *terms* like:

$$\mathtt{X}, \qquad \mathtt{1}, \qquad \mathtt{R}, \qquad \mathtt{Y}, \qquad \mathtt{R+(Y{\times}Q)}, \qquad \mathtt{Y{\times}Q}$$

Terms denote *values* such as numbers and strings, unlike statements which are either true or false. Some terms, like 1 and $\mathtt{4} + \mathtt{5}$, denote a fixed value, whilst other terms contain *variables* like X, Y, Z etc. whose value can vary. We will use conventional mathematical notation for terms, as illustrated by the examples below:

[5] See the IB course *Logic and Proof*.

$$\texttt{X},\quad \texttt{Y},\quad \texttt{Z},$$

$$\texttt{1},\quad \texttt{2},\quad \texttt{325},$$

$$\texttt{-X},\quad \texttt{-(X+1)},\quad \texttt{(X}\times\texttt{Y)+Z},$$

$$\sqrt{\texttt{(1+X}^2\texttt{)}},\quad \texttt{X!},\quad \texttt{sin(X)},\quad \texttt{rem(X,Y)}$$

`T` and `F` are atomic statements that are always true and false respectively. Other atomic statements are built from terms using *predicates*. Here are some more examples:

$$\texttt{ODD(X)},\quad \texttt{PRIME(3)},\quad \texttt{X} = \texttt{1},\quad \texttt{(X+1)}^2 \geq \texttt{X}^2$$

`ODD` and `PRIME` are examples of predicates and $=$ and $\geq$ are examples of *infixed* predicates. The expressions `X`, `1`, `3`, `X+1`, $\texttt{(X+1)}^2$, $\texttt{X}^2$ are examples of terms.

Compound statements are built up from atomic statements using the following logical operators:

$\neg$	(not)
$\wedge$	(and)
$\vee$	(or)
$\Rightarrow$	(implies)
$\Leftrightarrow$	(if and only if)

The single arrow $\rightarrow$ is commonly used for implication instead of $\Rightarrow$. We use $\Rightarrow$ to avoid possible confusion with the the use of $\rightarrow$ for λ-conversion in Part II.

Suppose P and Q are statements, then:

- $\neg P$ is true if P is false, and false if P is true.
- $P \wedge Q$ is true whenever both P and Q are true.
- $P \vee Q$ is true if either P or Q (or both) are true.
- $P \Rightarrow Q$ is true if whenever P is true, then Q is true also. By convention we regard $P \Rightarrow Q$ as being true if P is false. In fact, it is common to regard $P \Rightarrow Q$ as equivalent to $\neg P \vee Q$; however, some philosophers called intuitionists disagree with this treatment of implication.
- $P \Leftrightarrow Q$ is true if P and Q are either both true or both false. In fact $P \Leftrightarrow Q$ is equivalent to $(P \Rightarrow Q) \wedge (Q \Rightarrow P)$.

Examples of statements built using the connectives are:

$\texttt{ODD(X)} \vee \texttt{EVEN(X)}$	`X` is odd or even.
$\neg(\texttt{PRIME(X)} \Rightarrow \texttt{ODD(X)})$	It is not the case that if `X` is prime, then `X` is odd.
$\texttt{X} \leq \texttt{Y} \Rightarrow \texttt{X} \leq \texttt{Y}^2$	If `X` is less than or equal to `Y`, then `X` is less than or equal to $\texttt{Y}^2$.

To reduce the need for brackets it is assumed that $\neg$ is more binding than $\wedge$ and $\vee$, which in turn are more binding than $\Rightarrow$ and $\Leftrightarrow$. For example:

$\neg P \wedge Q$	is equivalent to	$(\neg P) \wedge Q$
$P \wedge Q \Rightarrow R$	is equivalent to	$(P \wedge Q) \Rightarrow R$
$P \wedge Q \Leftrightarrow \neg R \vee S$	is equivalent to	$(P \wedge Q) \Leftrightarrow ((\neg R) \vee S)$

Chapter 2

Floyd-Hoare Logic

The idea of formal proof is discussed. Floyd-Hoare logic is then introduced as a method for reasoning formally about programs.

In the last chapter three kinds of expressions that could be true or false were introduced:

(i) Partial correctness specifications $\{P\}\ C\ \{Q\}$.

(ii) Total correctness specifications $[P]\ C\ [Q]$.

(iii) Statements of mathematics (e.g. $(\mathtt{X}+\mathtt{1})^2 = \mathtt{X}^2 + \mathtt{2}\times\mathtt{X}+\mathtt{1}$).

It is assumed that the reader knows how to prove simple mathematical statements like the one in (iii) above. Here, for example, is a proof of this fact.

1.	$(\mathtt{X}+\mathtt{1})^2$	$= (\mathtt{X}+\mathtt{1})\times(\mathtt{X}+\mathtt{1})$	Definition of $()^2$.
2.	$(\mathtt{X}+\mathtt{1})\times(\mathtt{X}+\mathtt{1})$	$= (\mathtt{X}+\mathtt{1})\times\mathtt{X} + (\mathtt{X}+\mathtt{1})\times\mathtt{1}$	Left distributive law of $\times$ over $+$.
3.	$(\mathtt{X}+\mathtt{1})^2$	$= (\mathtt{X}+\mathtt{1})\times\mathtt{X} + (\mathtt{X}+\mathtt{1})\times\mathtt{1}$	Substituting line 2 into line 1.
4.	$(\mathtt{X}+\mathtt{1})\times\mathtt{1}$	$= \mathtt{X}+\mathtt{1}$	Identity law for $\mathtt{1}$.
5.	$(\mathtt{X}+\mathtt{1})\times\mathtt{X}$	$= \mathtt{X}\times\mathtt{X}+\mathtt{1}\times\mathtt{X}$	Right distributive law of $\times$ over $+$.
6.	$(\mathtt{X}+\mathtt{1})^2$	$= \mathtt{X}\times\mathtt{X}+\mathtt{1}\times\mathtt{X}+\mathtt{X}+\mathtt{1}$	Substituting lines 4 and 5 into line 3.
7.	$\mathtt{1}\times\mathtt{X}$	$= \mathtt{X}$	Identity law for $\mathtt{1}$.
8.	$(\mathtt{X}+\mathtt{1})^2$	$= \mathtt{X}\times\mathtt{X}+\mathtt{X}+\mathtt{X}+\mathtt{1}$	Substituting line 7 into line 6.
9.	$\mathtt{X}\times\mathtt{X}$	$= \mathtt{X}^2$	Definition of $()^2$.
10.	$\mathtt{X}+\mathtt{X}$	$= \mathtt{2}\times\mathtt{X}$	2=1+1, distributive law.
11.	$(\mathtt{X}+\mathtt{1})^2$	$= \mathtt{X}^2+\mathtt{2}\times\mathtt{X}+\mathtt{1}$	Substituting lines 9 and 10 into line 8.

This proof consists of a sequence of lines, each of which is an instance of an *axiom* (like the definition of $()^2$) or follows from previous lines by a *rule of inference* (like the substitution of equals for equals). The statement occurring on the last line of a proof is the statement *proved* by it (thus $(\mathtt{X}+\mathtt{1})^2 = \mathtt{X}^2+\mathtt{2}\times\mathtt{X}+\mathtt{1}$ is proved by the proof above).

To construct formal proofs of partial correctness specifications axioms and rules of inference are needed. This is what Floyd-Hoare logic provides. The formulation of the deductive system is due to Hoare [20], but some of the underlying ideas originated with Floyd [11].

A proof in Floyd-Hoare logic is a sequence of lines, each of which is either an *axiom* of the logic or follows from earlier lines by a *rule of inference* of the logic.

The reason for constructing formal proofs is to try to ensure that only sound methods of deduction are used. With sound axioms and rules of inference, one can be confident that the conclusions are true. On the other hand, if any axioms or rules of inference are unsound then it may be possible to deduce false conclusions; for example[1]

1.	$\sqrt{-1 \times -1}$	$= \sqrt{-1 \times -1}$	Reflexivity of $=$.
2.	$\sqrt{-1 \times -1}$	$= (\sqrt{-1}) \times (\sqrt{-1})$	Distributive law of $\sqrt{\ }$ over $\times$.
3.	$\sqrt{-1 \times -1}$	$= (\sqrt{-1})^2$	Definition of $()^2$.
4.	$\sqrt{-1 \times -1}$	$= -1$	definition of $\sqrt{\ }$.
5.	$\sqrt{1}$	$= -1$	As $-1 \times -1 = 1$.
6.	1	$= -1$	As $\sqrt{1} = 1$.

A formal proof makes explicit what axioms and rules of inference are used to arrive at a conclusion. It is quite easy to come up with plausible rules for reasoning about programs that are actually unsound (some examples for `FOR`-commands can be found in Section 2.1.11). Proofs of correctness of computer programs are often very intricate and formal methods are needed to ensure that they are valid. It is thus important to make fully explicit the reasoning principles being used, so that their soundness can be analysed.

Exercise 6
Find the flaw in the 'proof' of $1 = -1$ above. $\Box$

For some applications, correctness is especially important. Examples include life-critical systems such as nuclear reactor controllers, car braking systems, fly-by-wire aircraft and software controlled medical equipment. At the time of writing, there is a legal action in progress resulting from the death of several people due to radiation overdoses by a cancer treatment machine that had a software bug [25]. Formal proof of correctness provides a way of establishing the absence of bugs when exhaustive testing is impossible (as it almost always is).

The Floyd-Hoare deductive system for reasoning about programs will be explained and illustrated, but the mathematical analysis of the soundness and completeness of the system is only briefly discussed (see Section 2.2).

2.1 Axioms and rules of Floyd-Hoare logic

As discussed at the beginning of this chapter, a *formal proof* of a statement is a sequence of lines ending with the statement and such that each line is either an instance of an axiom or follows from previous lines by a rule of inference. If S is a statement (of either ordinary mathematics or Floyd-Hoare logic) then we write $\vdash S$ to mean that S has a proof. The statements that have proofs are called *theorems*.

[1] This example was shown to me by Sylva Cohn.

As discussed earlier, in these notes only the axioms and rules of inference for Floyd-Hoare logic are described; we will thus simply assert $\vdash S$ if S is a theorem of mathematics without giving any formal justification. Of course, to achieve complete rigour such assertions must be proved, but for details of how to do this the reader will have to consult the later chapters on first order and higher order logic.

The axioms of Floyd-Hoare logic are specified below by *schemas* which can be *instantiated* to get particular partial correctness specifications. The inference rules of Floyd-Hoare logic will be specified with a notation of the form:

$$\frac{\vdash\ S_1,\ \ldots\ ,\ \vdash\ S_n}{\vdash\ S}$$

This means the *conclusion* $\vdash S$ may be deduced from the *hypotheses* $\vdash S_1$, ... , $\vdash S_n$. The hypotheses can either all be theorems of Floyd-Hoare logic (as in the sequencing rule below), or a mixture of theorems of Floyd-Hoare logic and theorems of mathematics (as in the rule of preconditioning strengthening described in Section 2.1.2).

2.1.1 The assignment axiom

The assignment axiom represents the fact that the value of a variable V *after* executing an assignment command V`:=`E equals the value of the expression E in the state *before* executing it. To formalize this, observe that if a statement P is to be true *after* the assignment, then the statement obtained by substituting E for V in P must be true *before* executing it.

In order to say this formally, define $P\texttt{[}E/V\texttt{]}$ to mean the result of replacing all occurrences of V in P by E. Read $P\texttt{[}E/V\texttt{]}$ as 'P with E for V'. For example,

$$(\texttt{X}+1 > \texttt{X})\texttt{[Y+Z/X]} \ = \ ((\texttt{Y+Z})+1 > \texttt{Y+Z})$$

The way to remember this notation is to remember the 'cancellation law'

$$V\texttt{[}E/V\texttt{]} \ = \ E$$

which is analogous to the cancellation property of fractions

$$v \times (e/v) \ = \ e$$

The assignment axiom

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

Where V is any variable, E is any expression, P is any statement and the notation $P\texttt{[}E/V\texttt{]}$ denotes the result of substituting the term E for all occurrences of the variable V in the statement P.

Instances of the assignment axiom are:

1. $\vdash$ $\{\mathtt{Y} = \mathtt{2}\}$ $\mathtt{X} := \mathtt{2}$ $\{\mathtt{Y} = \mathtt{X}\}$
2. $\vdash$ $\{\mathtt{X} + \mathtt{1} = \mathtt{n} + \mathtt{1}\}$ $\mathtt{X} := \mathtt{X} + \mathtt{1}$ $\{\mathtt{X} = \mathtt{n} + \mathtt{1}\}$
3. $\vdash$ $\{E = E\}$ $\mathtt{X} := E$ $\{\mathtt{X} = E\}$ (if `X` does not occur in E).

Many people feel the assignment axiom is 'backwards' from what they would expect. Two common erroneous intuitions are that it should be as follows:

(i) $\vdash$ $\{P\}$ V:=E $\{P[V/E]\}$.

Where the notation $P[V/E]$ denotes the result of substituting V for E in P.

This has the clearly false consequence that $\vdash$ $\{$X=0$\}$ X:=1 $\{$X=0$\}$, since the (X=0)[X/1] is equal to (X=0) as 1 doesn't occur in (X=0).

(ii) $\vdash$ $\{P\}$ V:=E $\{P[E/V]\}$.

This has the clearly false consequence $\vdash$ $\{$X=0$\}$ X:=1 $\{$1=0$\}$ which follows by taking P to be X=0, V to be X and E to be 1.

The fact that it is easy to have wrong intuitions about the assignment axiom shows that it is important to have rigorous means of establishing the validity of axioms and rules. We will not go into this topic here aside from remarking that it is possible to give a *formal semantics* of our little programming language and then to *prove* that the axioms and rules of inference of Floyd-Hoare logic are sound. Of course, this process will only increase our confidence in the axioms and rules to the extent that we believe the correctness of the formal semantics. The simple assignment axiom above is not valid for 'real' programming languages. For example, work by G. Ligler [27] shows that it can fail to hold in six different ways for the language Algol 60.

One way that our little programming language differs from real languages is that the evaluation of expressions on the right of assignment commands cannot 'side effect' the state. The validity of the assignment axiom depends on this property. To see this, suppose that our language were extended so that it contained the 'block expression'

```
BEGIN Y:=1; 2 END
```

This expression, E say, has value 2, but its evaluation also 'side effects' the variable Y by storing 1 in it. If the assignment axiom applied to expressions like E, then it could be used to deduce:

$\vdash$ $\{$Y=0$\}$ X:=BEGIN Y:=1; 2 END $\{$Y=0$\}$

(since (Y=0)[E/X] = (Y=0) as X does not occur in (Y=0)). This is clearly false, as after the assignment Y will have the value 1.

Floyd-Hoare logic can be extended to cope with arrays so that, for example, the correctness of inplace sorting programs can be verified. However, it is not as straightforward as one might expect to do this. The main problem is that the assignment axiom does not apply to array assignments of the form $A(E_1)$:=E_2 (where A is an array variable and E_1 and E_2 are expressions).

One might think that the assignment axiom in could be generalized to

$$\vdash \ \{P\texttt{[}E_2/A(E_1)\texttt{]}\} \ A(E_1) := E_2 \ \{P\}$$

where '$P\texttt{[}E_2/A(E_1)\texttt{]}$' denotes the result of substituting E_2 for all occurrences of $A(E_1)$ throughout P. Alas, this does not work. Consider the following case:

$$P \ \equiv \text{`}\texttt{A(Y)=0}\text{'}, \qquad E_1 \ \equiv \text{`}\texttt{X}\text{'}, \qquad E_2 \ \equiv \text{`}\texttt{1}\text{'}$$

Since `A(X)` does not occur in P, it follows that P`[1/A(X)] =` P, and hence the generalized axiom yields

$$\vdash \{\texttt{A(Y)=0}\} \ \texttt{A(X):=1} \ \{\texttt{A(Y)=0}\}$$

This specification is clearly false if `X=Y`. To avoid this, the array assignment axiom must take into account the possibility that changes to `A(X)` may also change `A(Y)`, `A(Z)`, ... (since `X` might equal `Y`, `Z`, ...). This is discussed further in Section 2.1.12.

2.1.2 Precondition strengthening

The next rule of Floyd-Hoare logic enables the preconditions of (i) and (ii) on page 20 to be simplified. Recall that

$$\frac{\vdash \ S_1, \ \ldots \ , \ \vdash \ S_n}{\vdash \ S}$$

means that $\vdash \ S$ can be deduced from $\vdash \ S_1, \ldots, \vdash \ S_n$.

Using this notation, the rule of precondition strengthening is

Precondition strengthening

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

Examples

1. From the arithmetic fact $\vdash$ `X=n` $\Rightarrow$ `X+1=n+1`, and 2 on page 20 it follows by precondition strengthening that

$$\vdash \ \{\texttt{X} = \texttt{n}\} \ \texttt{X} := \texttt{X} + \texttt{1} \ \{\texttt{X} = \texttt{n} + \texttt{1}\}.$$

The variable `n` is an example of an *auxiliary* (or *ghost*) variable. As described earlier (see page 13), auxiliary variables are variables occurring in a partial correctness specification $\{P\}\ C\ \{Q\}$ which do not occur in the command C. Such variables are used to relate values in the state before and after C is executed. For example, the specification above says that if the value of $\mathtt{X}$ is $\mathtt{n}$, then after executing the assignment $\mathtt{X:=X+1}$ its value will be $\mathtt{n+1}$.

2. From the logical truth $\vdash\ \mathtt{T}\ \Rightarrow\ (E\mathtt{=}E)$, and 3 on page 20 one can deduce that if $\mathtt{X}$ is not in E then:

$$\vdash\ \{\mathtt{T}\}\ \mathtt{X}:=E\ \{\mathtt{X}=E\}$$

□

2.1.3 Postcondition weakening

Just as the previous rule allows the precondition of a partial correctness specification to be strengthened, the following one allows us to weaken the postcondition.

Postcondition weakening

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

Example: Here is a little formal proof.

1.	$\vdash\ \{\mathtt{R=X} \wedge \mathtt{0=0}\}\ \mathtt{Q:=0}\ \{\mathtt{R=X} \wedge \mathtt{Q=0}\}$	By the assignment axiom.
2.	$\vdash\ \mathtt{R=X}\ \Rightarrow\ \mathtt{R=X} \wedge \mathtt{0=0}$	By pure logic.
3.	$\vdash\ \{\mathtt{R=X}\}\ \mathtt{Q=0}\ \{\mathtt{R=X} \wedge \mathtt{Q=0}\}$	By precondition strengthening.
4.	$\vdash\ \mathtt{R=X} \wedge \mathtt{Q=0}\ \Rightarrow\ \mathtt{R=X+(Y} \times \mathtt{Q)}$	By laws of arithmetic.
5.	$\vdash\ \{\mathtt{R=X}\}\ \mathtt{Q:=0}\ \{\mathtt{R=X+(Y} \times \mathtt{Q)}\}$	By postcondition weakening.

□

The rules precondition strengthening and postcondition weakening are sometimes called the *rules of consequence*.

2.1.4 Specification conjunction and disjunction

The following two rules provide a method of combining different specifications about the same command.

Specification conjunction

$$\frac{\vdash \{P_1\}\ C\ \{Q_1\}, \qquad \vdash \{P_2\}\ C\ \{Q_2\}}{\vdash \{P_1 \wedge P_2\}\ C\ \{Q_1 \wedge Q_2\}}$$

Specification disjunction

$$\frac{\vdash \{P_1\}\ C\ \{Q_1\}, \qquad \vdash \{P_2\}\ C\ \{Q_2\}}{\vdash \{P_1 \vee P_2\}\ C\ \{Q_1 \vee Q_2\}}$$

These rules are useful for splitting a proof into independent bits. For example, they enable $\vdash$ {P} C {$\mathtt{Q}_1 \wedge \mathtt{Q}_2$} to be proved by proving separately that both $\vdash$ {P} C {$\mathtt{Q}_1$} and $\vdash$ {P} C {$\mathtt{Q}_2$}.

The rest of the rules allow the deduction of properties of compound commands from properties of their components.

2.1.5 The sequencing rule

The next rule enables a partial correctness specification for a sequence $C_1;C_2$ to be derived from specifications for C_1 and C_2.

The sequencing rule

$$\frac{\vdash \{P\}\ C_1\ \{Q\}, \qquad \vdash \{Q\}\ C_2\ \{R\}}{\vdash \{P\}\ C_1;C_2\ \{R\}}$$

Example: By the assignment axiom:

(i) $\vdash$ {X=x∧Y=y} R:=X {R=x∧Y=y}

(ii) $\vdash$ {R=x∧Y=y} X:=Y {R=x∧X=y}

(iii) $\vdash$ {R=x∧X=y} Y:=R {Y=x∧X=y}

Hence by (i), (ii) and the sequencing rule

(iv) $\vdash$ {X=x∧Y=y} R:=X; X:=Y {R=x∧X=y}

Hence by (iv) and (iii) and the sequencing rule

(v) $\vdash$ {X=x∧Y=y} R:=X; X:=Y; Y:=R {Y=x∧X=y}

□

2.1.6 The derived sequencing rule

The following rule is derivable from the sequencing and consequence rules.

The derived sequencing rule

$$\frac{\begin{array}{ll} & \vdash\ P \Rightarrow P_1 \\ \vdash\ \{P_1\}\ C_1\ \{Q_1\} & \vdash\ Q_1 \Rightarrow P_2 \\ \vdash\ \{P_2\}\ C_2\ \{Q_2\} & \vdash\ Q_2 \Rightarrow P_3 \\ . & . \\ . & . \\ . & . \\ \vdash\ \{P_n\}\ C_n\ \{Q_n\} & \vdash\ Q_n \Rightarrow Q \end{array}}{\vdash\ \{P\}\ C_1;\ \ldots\ ;\ C_n\ \{Q\}}$$

The derived sequencing rule enables (v) in the previous example to be deduced directly from (i), (ii) and (iii) in one step.

2.1.7 The block rule

The block rule is like the sequencing rule, but it also takes care of local variables.

The block rule

$$\frac{\vdash\ \{P\}\ C\ \{Q\}}{\vdash\ \{P\}\ \texttt{BEGIN VAR}\ V_1\texttt{;}\ \ldots\texttt{;}\ \texttt{VAR}\ V_n\texttt{;}\ C\ \texttt{END}\ \{Q\}}$$

where none of the variables $V_1, \ldots, V_n$ occur in P or Q.

The syntactic condition that none of the variables $V_1, \ldots, V_n$ occur in P or Q is an example of a *side condition*. It is a syntactic condition that must hold whenever the rule is used. Without this condition the rule is invalid; this is illustrated in the example below.

Note that the block rule is regarded as including the case when there are no local variables (the '$n = 0$' case).

Example: From $\vdash$ {`X=x` ∧ `Y=y`} `R:=X; X:=Y; Y:=R` {`Y=x` ∧ `X=y`} (see page 23) it follows by the block rule that

$$\vdash\ \{\texttt{X=x}\ \wedge\ \texttt{Y=y}\}\ \texttt{BEGIN VAR R; R:=X; X:=Y; Y:=R END}\ \{\texttt{Y=x}\ \wedge\ \texttt{X=y}\}$$

since `R` does not occur in `X=x` ∧ `Y=y` or `X=y` ∧ `Y=x`. Notice that from

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

one *cannot* deduce

$$\vdash \{\texttt{X=x} \wedge \texttt{Y=y}\}\ \texttt{BEGIN VAR R; R:=X; X:=Y END}\ \{\texttt{R=x} \wedge \texttt{X=y}\}$$

since `R` occurs in $\{\texttt{R=x} \wedge \texttt{X=y}\}$. This is as required, because assignments to local variables of blocks should not be felt outside the block body. Notice, however, that it is possible to deduce:

$$\vdash \{\texttt{X=x} \wedge \texttt{Y=y}\}\ \texttt{BEGIN R:=X; X:=Y END}\ \{\texttt{R=x} \wedge \texttt{X=y}\}.$$

This is correct because `R` is no longer a local variable. □

The following exercise addresses the question of whether one can show that changes to local variables inside a block are invisible outside it.

Exercise 7

Consider the specification

$$\{\texttt{X=x}\}\ \texttt{BEGIN VAR X; X:=1 END}\ \{\texttt{X=x}\}$$

Can this be deduced from the rules given so far?

(i) If so, give a proof of it.

(ii) If not, explain why not and suggest additional rules and/or axioms to enable it to be deduced.

□

2.1.8 The derived block rule

From the derived sequencing rule and the block rule the following rule for blocks can be derived.

The derived block rule

$$\frac{\begin{array}{ll} & \vdash\ P \Rightarrow P_1 \\ \vdash\ \{P_1\}\ C_1\ \{Q_1\} & \vdash\ Q_1 \Rightarrow P_2 \\ \vdash\ \{P_2\}\ C_2\ \{Q_2\} & \vdash\ Q_2 \Rightarrow P_3 \\ . & . \\ . & . \\ . & . \\ \vdash\ \{P_n\}\ C_n\ \{Q_n\} & \vdash\ Q_n \Rightarrow Q \end{array}}{\vdash\ \{P\}\ \texttt{BEGIN VAR}\ V_1\texttt{;}\ \ldots\ \texttt{VAR}\ V_m\texttt{;}C_1\texttt{;}\ \ldots\ \texttt{;}\ C_n\ \{Q\}}$$

where none of the variables $V_1, \ldots, V_m$ occur in P or Q.

Using this rule, it can be deduced *in one step* from (i), (ii) and (iii) on page 23 that:

$$\vdash\ \{\texttt{X=x} \wedge \texttt{Y=y}\}\ \texttt{BEGIN VAR R; R:=X; X:=Y; Y:=R END}\ \{\texttt{Y=x} \wedge \texttt{X=y}\}$$

Exercise 8

Is the following specification true?

$\vdash$ {`X=x` $\wedge$ `Y=y`} `X:=X+Y; Y:=X-Y; X:=X-Y` {`Y=x` $\wedge$ `X=y`}

If so, prove it. If not, give the circumstances in which it fails. □

Exercise 9

Show $\vdash$ {`X=R+(Y`$\times$`Q)`} `BEGIN R:=R-Y; Q:=Q+1 END` {`X=R+(Y`$\times$`Q)`}

□

2.1.9 The conditional rules

There are two kinds of conditional commands: one-armed conditionals and two-armed conditionals. There are thus two rules for conditionals.

The conditional rules

$$\frac{\vdash\ \{P \wedge S\}\ C\ \{Q\}, \qquad \vdash\ P \wedge \neg S \Rightarrow Q}{\vdash\ \{P\}\ \texttt{IF}\ S\ \texttt{THEN}\ C\ \{Q\}}$$

$$\frac{\vdash\ \{P \wedge S\}\ C_1\ \{Q\}, \qquad \vdash\ \{P \wedge \neg S\}\ C_2\ \{Q\}}{\vdash\ \{P\}\ \texttt{IF}\ S\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2\ \{Q\}}$$

Example: Suppose we are given that

(i) $\vdash$ `X`$\geq$`Y` $\Rightarrow$ `max(X,Y)=X`

(ii) $\vdash$ `Y`$\geq$`X` $\Rightarrow$ `max(X,Y)=Y`

Then by the conditional rules (and others) it follows that

$\vdash$ {`T`} `IF X`$\geq$`Y THEN MAX:=X ELSE MAX:=Y` {`MAX=max(X,Y)`}

□

Exercise 10

Give a detailed formal proof that the specification in the previous example follows from hypotheses (i) and (ii). □

Exercise 11

Devise an axiom and/or rule of inference for a command `SKIP` that has no effect. Show that if `IF` S `THEN` C is regarded as an abbreviation for `IF` S `THEN` C `ELSE SKIP`, then the rule for one-armed conditionals is derivable from the rule for two-armed conditionals and your axiom/rule for `SKIP`. □

Exercise 12

Suppose we add to our little programming language commands of the form:

$$\texttt{CASE}\ E\ \texttt{OF BEGIN}\ C_1;\ \ldots\ ;\ C_n\ \texttt{END}$$

These are evaluated as follows:

(i) First E is evaluated to get a value x.

(ii) If x is not a number between 1 and n, then the `CASE`-command has no effect.

(iii) If $x = i$ where $1 \leq i \leq n$, then command C_i is executed.

Why is the following rule for `CASE`-commands wrong?

$$\frac{\vdash\ \{P\ \wedge\ E=1\}\ C_1\ \{Q\},\ \ \ldots\ \ ,\vdash\ \{P\ \wedge\ E=n\}\ C_n\ \{Q\}}{\vdash\ \{P\}\ \texttt{CASE}\ E\ \texttt{OF BEGIN}\ C_1;\ \ldots\ ;\ C_n\ \texttt{END}\ \{Q\}}$$

Hint: Consider the case when P is '$X = 0$', E is 'X', C_1 is 'Y:=0' and Q is '$Y = 0$'.

□

Exercise 13

Devise a proof rule for the `CASE`-commands in the previous exercise and use it to show:

```
⊢ {1≤X∧X≤3}
 CASE X OF
   BEGIN
     Y:=X-1;
     Y:=X-2;
     Y:=X-3
   END
   {Y=0}
```

□

Exercise 14

Show that if $\vdash$ {P∧S} $\texttt{C}_1$ {Q} and $\vdash$ {P∧¬S} $\texttt{C}_2$ {Q}, then it is possible to deduce:

$$\vdash\ \{\texttt{P}\}\ \texttt{IF S THEN C}_1\ \texttt{ELSE IF}\ \neg\texttt{S THEN C}_2\ \{\texttt{Q}\}.$$

□

2.1.10 The `WHILE`-rule

If $\vdash \{P \wedge S\}\ C\ \{P\}$, we say: P is an *invariant* of C whenever S holds. The `WHILE`-rule says that if P is an invariant of the body of a `WHILE`-command whenever the test condition holds, then P is an invariant of the whole `WHILE`-command. In other words, if executing C once preserves the truth of P, then executing C any number of times also preserves the truth of P.

The `WHILE`-rule also expresses the fact that after a `WHILE`-command has terminated, the test must be false (otherwise, it wouldn't have terminated).

The `WHILE`-rule

$$\frac{\vdash\ \{P \wedge S\}\ C\ \{P\}}{\vdash\ \{P\}\ \texttt{WHILE}\ S\ \texttt{DO}\ C\ \{P \wedge \neg S\}}$$

Example: By Exercise 9 on page 26

$$\vdash \{\texttt{X=R+(Y}\times\texttt{Q)}\}\ \texttt{BEGIN R:=R-Y; Q:=Q+1 END}\ \{\texttt{X=R+(Y}\times\texttt{Q)}\}$$

Hence by precondition strengthening

$$\vdash \{\texttt{X=R+(Y}\times\texttt{Q)}\wedge\texttt{Y}\leq\texttt{R}\}\ \texttt{BEGIN R:=R-Y; Q:=Q+1 END}\ \{\texttt{X=R+(Y}\times\texttt{Q)}\}$$

Hence by the `WHILE`-rule (with P = '`X=R+(Y×Q)`')

```
(i)  ⊢ {X=R+(Y×Q)}
      WHILE Y≤R DO
        BEGIN R:=R-Y; Q:=Q+1 END
     {X=R+(Y×Q)∧¬(Y≤R)}
```

It is easy to deduce that

```
(ii)  ⊢ {T} R:=X; Q:=0 {X=R+(Y×Q)}
```

Hence by (i) and (ii), the sequencing rule and postcondition weakening

```
⊢ {T}
 R:=X;
 Q:=0;
 WHILE Y≤R DO
   BEGIN R:=R-Y; Q:=Q+1 END
{R<Y∧X=R+(Y×Q)}
```

□

With the exception of the `WHILE`-rule, all the axioms and rules described so far are sound for total correctness as well as partial correctness. This is because the only commands in our little language that might not terminate are `WHILE`-commands. Consider now the following proof:

1. $\vdash$ {T} X:=0 {T} (assignment axiom)
2. $\vdash$ {T $\wedge$ T} X:=0 {T} (precondition strengthening)
3. $\vdash$ {T} WHILE T DO X:=0 {T $\wedge$ $\neg$T} (2 and the WHILE-rule)

If the WHILE-rule were true for total correctness, then the proof above would show that:

$$\vdash \ [\mathtt{T}]\ \mathtt{WHILE\ T\ DO\ X{:=}0}\ [\mathtt{T} \wedge \neg\mathtt{T}]$$

but this is clearly false since WHILE T DO X:=0 does not terminate, and even if it did then T $\wedge$ $\neg$T could not hold in the resulting state.

Extending Floyd-Hoare logic to deal with termination is quite tricky. One approach can be found in Dijkstra [10].

2.1.11 The FOR-rule

It is quite hard to capture accurately the intended semantics of FOR-commands in Floyd-Hoare logic. Axioms and rules are given here that *appear* to be sound, but they are not necessarily complete (see Section 2.2). An early reference on the logic of FOR-commands is Hoare's 1972 paper [21]; a comprehensive treatment can be found in Reynolds [38].

The intention here in presenting the FOR-rule is to show that Floyd-Hoare logic can get very tricky. All the other axioms and rules were quite straightforward and may have given a false sense of simplicity: it is very difficult to give adequate rules for anything other than very simple programming constructs. This is an important incentive for using simple languages.

One problem with FOR-commands is that there are many subtly different versions of them. Thus before describing the FOR-rule, the intended semantics of FOR-commands must be described carefully. In these notes, the semantics of

$$\mathtt{FOR}\ V\mathtt{:=}E_1\ \mathtt{UNTIL}\ E_2\ \mathtt{DO}\ C$$

is as follows:

(i) The expressions E_1 and E_2 are evaluated once to get values e_1 and e_2, respectively.

(ii) If either e_1 or e_2 is not a number, or if $e_1 > e_2$, then nothing is done.

(iii) If $e_1 \leq e_2$ the FOR-command is equivalent to:

$$\begin{array}{l}\mathtt{BEGIN\ VAR}\ V\mathtt{;}\\ \quad V\mathtt{:=}e_1\mathtt{;}\ C\mathtt{;}\ V\mathtt{:=}e_1\mathtt{+}1\mathtt{;}\ C\ \mathtt{;}\ \ldots\ \mathtt{;}\ V\mathtt{:=}e_2\mathtt{;}\ C\\ \mathtt{END}\end{array}$$

i.e. C is executed $(e_2{-}e_1){+}1$ times with V taking on the sequence of values e_1, $e_1{+}1$, ... , e_2 in succession. Note that this description is not rigorous: 'e_1' and 'e_2' have been used both as numbers and as expressions of our little language; the semantics of FOR-commands should be clear despite this.

`FOR`-rules in different languages can differ in subtle ways from the one here. For example, the expressions E_1 and E_2 could be evaluated at each iteration and the controlled variable V could be treated as global rather than local. Note that with the semantics presented here, `FOR`-commands cannot go into infinite loops (unless, of course, they contain non-terminating `WHILE`-commands).

To see how the `FOR`-rule works, suppose that

$\vdash \{P\}\ C\ \{P[V{+}1/V]\}$

Suppose also that C does not contain any assignments to the variable V. If this is the case, then it is intuitively clear (and can be rigorously proved) that

$\vdash \{(V = v)\}\ C\ \{(V = v)\}$

hence by specification conjunction

$\vdash \{P \wedge (V = v)\}\ C\ \{P[V{+}1/V]\ \wedge (V = v)\}$

Now consider a sequence

V:=v; C.

By Example 2 on page 22,

$\vdash \{P[v/V]\}\ V{:=}v\ \{P \wedge (V = v)\}$

Hence by the sequencing rule

$\vdash \{P[v/V]\}\ V{:=}v;\ C\ \{P[V{+}1/V]\ \wedge (V = v)\}$

Now it is a truth of logic alone that

$\vdash\ P[V{+}1/V] \wedge (V = v)\ \Rightarrow\ P[v{+}1/V]$

hence by postcondition weakening

$\vdash \{P[v/V]\}\ V{:=}v;\ C\ \{P[v{+}1/V]\}$

Taking v to be e_1, $e_1{+}1$, $\ldots$, e_2

$\vdash \{P[e_1/V]\}\ V{:=}e_1;\ C\ \{P[e_1{+}1/V]\}$
$\vdash \{P[e_1{+}1/V]\}\ V{:=}e_1{+}1;\ C\ \{P[e_1{+}2/V]\}$
$\vdots$
$\vdash \{P[e_2/V]\}\ V{:=}e_2;\ C\ \{P[e_2{+}1/V]\}$

Hence by the derived sequencing rule:

$\{P[e_1/V]\}\ V{:=}e_1;\ C;\ V{:=}e_1{+}1;\ \ldots\ ;\ V{:=}e_2;\ C\ \{P[e_2{+}1/V]\}$

This suggests that a `FOR`-rule could be:

$$\frac{\vdash \{P\}\ C\ \{P[V\texttt{+1}/V]\}}{\vdash \{P[E_1/V]\}\ \texttt{FOR}\ V{:=}E_1\ \texttt{UNTIL}\ E_2\ \texttt{DO}\ C\ \{P[E_2\texttt{+1}/V]\}}$$

Unfortunately, this rule is unsound. To see this, first note that:

1. $\vdash$ {`Y+1=Y+1`} `X:=Y+1` {`X=Y+1`} (assignment axiom)
2. $\vdash$ {`T`} `X:=Y+1` {`X= Y+1`} (1 and precondition strengthening)
3. $\vdash$ `X=Y` $\Rightarrow$ `T` (logic: ‘anything implies true’)
4. $\vdash$ {`X=Y`} `X:=Y+1` {`X=Y+1`} (2 and precondition strengthening)

Thus if P is '`X=Y`' then:

$$\vdash \ \{P\}\ \texttt{X:=Y+1}\ \{P\texttt{[Y+1/Y]}\}$$

and so by the `FOR`-rule above, if we take V to be `Y`, E_1 to be 3 and E_2 to be `1`, then

$$\vdash \ \{\ \underbrace{\texttt{X=3}}_{P\texttt{[3/Y]}}\ \}\ \texttt{FOR Y:=3 UNTIL 1 DO X:=Y+1}\ \{\ \underbrace{\texttt{X=2}}_{P\texttt{[1+1/Y]}}\ \}$$

This is clearly false: it was specified that if the value of E_1 were greater than the value of E_2 then the `FOR`-command should have no effect, but in this example it changes the value of `X` from `3` to `2`.

To solve this problem, the `FOR`-rule can be modified to

$$\frac{\vdash \{P\}\ C\ \{P\texttt{[}V\texttt{+1/}V\texttt{]}\}}{\vdash \{P\texttt{[}E_1\texttt{/}V\texttt{]}\ \wedge\ E_1 \leq E_2\}\ \texttt{FOR}\ V\texttt{:=}E_1\ \texttt{UNTIL}\ E_2\ \texttt{DO}\ C\ \{P\texttt{[}E_2\texttt{+1/}V\texttt{]}\}}$$

If this rule is used on the example above all that can be deduced is

$$\vdash \ \{\texttt{X=3}\ \wedge\ \underbrace{\texttt{3} \leq \texttt{1}}_{\text{never true!}}\ \}\ \texttt{FOR Y:=3 UNTIL 1 DO X:=Y+1}\ \{\texttt{X=2}\}$$

This conclusion is harmless since it only asserts that `X` will be changed if the `FOR`-command is executed in an impossible starting state.

Unfortunately, there is still a bug in our `FOR`-rule. Suppose we take P to be '`Y=1`', then it is straightforward to show that:

$$\vdash \ \{\underbrace{\texttt{Y=1}}_{P}\}\ \texttt{Y:=Y-1}\ \{\ \underbrace{\texttt{Y+1=1}}_{P\texttt{[Y+1/Y]}}\ \}$$

so by our latest `FOR`-rule

$$\vdash \ \{\ \underbrace{\texttt{1=1}}_{P\texttt{[1/Y]}}\ \wedge\ \texttt{1} \leq \texttt{1}\}\ \texttt{FOR Y:=1 UNTIL 1 DO Y:=Y-1}\ \{\ \underbrace{\texttt{2=1}}_{P\texttt{[1+1/Y]}}\ \}$$

Whatever the command does, it doesn't lead to a state in which `2=1`. The problem is that the body of the `FOR`-command modifies the controlled variable. It is not surprising that this causes problems, since it was explicitly assumed that the body didn't modify the controlled variable when we motivated the `FOR`-rule. It turns out that problems also arise if any variables in the expressions E_1 and E_2 (which specify the upper and lower bounds) are modified. For example, taking P to be `Z=Y`, then it is straightforward to show

$$\vdash \ \{\underbrace{\texttt{Z=Y}}_{P}\}\ \texttt{Z:=Z+1}\ \{\ \underbrace{\texttt{Z=Y+1}}_{P\texttt{[Y+1/Y]}}\ \}$$

hence the rule allows us the following to be derived:

$$\vdash \ \{\ \underbrace{\texttt{Z=1}}_{P\texttt{[1/Y]}}\ \wedge\ \texttt{1} \leq \texttt{Z}\}\ \texttt{FOR Y:=1 UNTIL Z DO Z:=Z+1}\ \{\ \underbrace{\texttt{Z=Z+1}}_{P\texttt{[Z+1/Y]}}\ \}$$

This is clearly wrong as one can never have `Z=Z+1` (subtracting `Z` from both sides would give `0=1`). One might think that this is not a problem because the `FOR`-command would never terminate. In some languages this might be the case, but the semantics of our language were carefully defined in such a way that `FOR`-commands always terminate (see the beginning of this section).

To rule out the problems that arise when the controlled variable or variables in the bounds expressions, are changed by the body, we simply impose a side condition on the rule that stipulates that the rule cannot be used in these situations. A debugged rule is thus:

The `FOR`-rule

$$\frac{\vdash \{P \wedge (E_1 \leq V) \wedge (V \leq E_2)\}\ C\ \{P[V\texttt{+1}/V]\}}{\vdash \{P[E_1/V] \wedge (E_1 \leq E_2)\}\ \texttt{FOR}\ V := E_1\ \texttt{UNTIL}\ E_2\ \texttt{DO}\ C\ \{P[E_2\texttt{+1}/V]\}}$$

where neither V, nor any variable occurring in E_1 or E_2, is assigned to in the command C.

This rule does not enable anything to be deduced about `FOR`-commands whose body assigns to variables in the bounds expressions. This precludes such assignments being used if commands are to be reasoned about. The strategy of only defining rules of inference for non-tricky uses of constructs helps ensure that programs are written in a perspicuous manner. It is possible to devise a rule that does cope with assignments to variables in bounds expressions, but it is not clear whether it is a good idea to have such a rule.

The `FOR`-axiom

To cover the case when $E_2 < E_1$, we need the `FOR`-axiom below.

The `FOR`-axiom

$$\vdash\ \{P \wedge (E_2 < E_1)\}\ \texttt{FOR}\ V := E_1\ \texttt{UNTIL}\ E_2\ \texttt{DO}\ C\ \{P\}$$

This says that when E_2 is less than E_1 the `FOR`-command has no effect.

Example: By the assignment axiom and precondition strengthening

$$\vdash\ \{\texttt{X = ((N-1)}\times\texttt{N) DIV 2}\}\ \texttt{X:=X+N}\ \{\texttt{X=(N}\times\texttt{(N+1)) DIV 2}\}$$

Strengthening the precondition of this again yields

$$\vdash\ \{\texttt{(X=((N-1}\times\texttt{N) DIV 2)}\wedge\texttt{(1}\leq\texttt{N)}\wedge\texttt{(N}\leq\texttt{M)}\}\ \texttt{X:=X+N}\ \{\texttt{X=(N}\times\texttt{(N+1)) DIV 2}\}$$

Hence by the `FOR`-rule

$\vdash$ $\{$`(X=((1-1)`$\times$`1) DIV 2)`$\wedge$`(1`$\leq$`M)`$\}$
`FOR N:=1 UNTIL M DO X:=X+N`
$\{$`X=(M`$\times$`(M+1)) DIV 2`$\}$

Hence

$\vdash$ $\{$`(X=0)`$\wedge$`(1`$\leq$`M)`$\}$ `FOR N:=1 UNTIL M DO X:=X+N` $\{$`X=(M`$\times$`(M+1)) DIV 2`$\}$

□

Note that if

(i) $\vdash\ \{P\}\ C\ \{P[V{+}1/V]\}$, or

(ii) $\vdash\ \{P \wedge (E_1 \leq V)\}\ C\ \{P[V{+}1/V]\}$, or

(iii) $\vdash\ \{P \wedge (V \leq E_2)\}\ C\ \{P[V{+}1/V]\}$

then by precondition strengthening one can infer

$$\vdash\ \{P \wedge (E_1 \leq V) \wedge (V \leq E_2)\}\ \ C\ \ \{P[V{+}1/V]\}$$

The separate `FOR`-rule and `FOR`-axiom are a bit clunky. A nice treatment suggested by John Wickerson is the following:

Wickerson's `FOR`-rule

$$\frac{\vdash P \Rightarrow R[E_1/V], \quad \vdash R \wedge V{>}E_2 \Rightarrow Q, \quad \vdash \{R \wedge V{\leq}E_2\}\ C\ \{R[V{+}1/V]\}}{\vdash \{P\}\ \texttt{FOR}\ V := E_1\ \texttt{UNTIL}\ E_2\ \texttt{DO}\ C\ \{Q\}}$$

where neither V, nor any variable occurring in E_1 or E_2, is assigned to in the command C.

Exercise 15

Derive this rule by applying the block, sequencing and `WHILE` rules to:

`BEGIN VAR` V`;` V`:=`E_1`;` `WHILE` $V{\leq}E_2$ `DO` $(C$`;` V`:=`$V{+}1)$ `END`

□

Exercise 16

Show that the `FOR`-axiom can be derived from Wickerson's rule (hint take R to be $P \wedge E_2 < V$). □

Yet another alternative `FOR`-rule has been suggested by Bob Tennent:

Tennent's `FOR`-rule

$$\frac{\vdash \{P[V{-}1/V] \wedge (E_1 \leq V) \wedge (V \leq E_2)\}\ C\ \{P\}}{\vdash \{P[E_1{-}1/V]{\wedge}(E_1{-}1{\leq}E_2)\}\ \texttt{FOR}\ V := E_1\ \texttt{UNTIL}\ E_2\ \texttt{DO}\ C\ \{P[E_2/V]\}}$$

where neither V, nor any variable occurring in E_1 or E_2, is assigned to in the command C.

This rule also has the property that the "special case" of executing the loop body 0 times can normally be handled without use of the FOR-axiom. Justify this claim.

Exercise 17
Justify Tennent's rule.
□

Exercise 18
Compare and contrast Wickerson's rule with Tennent's rule. Which do you prefer?
□

Exercise 19
Show that

```
⊢ {M≥1}
  BEGIN
   X:=0;
   FOR N:=1 UNTIL M DO X:=X+N
  END
   {X=(M×(M+1)) DIV 2}
```

□

It is clear from the discussion above that there are various options for reasoning about `FOR`-commands in Floyd-Hoare logic. It may well be that one could argue for a 'best' approach (though, as far as I know, there is no consensus on this for our toy language, which is not surprising as `FOR` loops in real languages are more complex). The point is that designing rules for constructs that go beyond the simple core language of assignment, sequencing, conditionals and `WHILE`-loops is tricky and may involve personal preferences.

2.1.12 Arrays

At the end of Section 2.1.1 it is shown that the naive array assignment axiom

$$\vdash\ \ \{P\texttt{[}E_2/A(E_1)\texttt{]}\}\ \ A(E_1) := E_2\ \ \{P\}$$

does not work, because of the possibility that changes to $A(X)$ may also change $A(Y)$, $A(Z)$, ... (since X might equal Y, Z, ...).

The solution, due to Hoare, is to treat an array assignment

$$A(E_1){:=}E_2$$

as an ordinary assignment

$$A := A\{E_1{\leftarrow}E_2\}$$

where the term $A\{E_1{\leftarrow}E_2\}$ denotes an array identical to A, except that the E_1-th component is changed to have the value E_2.

Thus an array assignment is just a special case of an ordinary variable assignment.

The array assignment axiom

$$\vdash \ \{P\texttt{[}A\{E_1{\leftarrow}E_2\}/A\texttt{]}\}\ A(E_1)\texttt{:=}E_2\ \{P\}$$

Where A is an array variable, E_1 is an integer valued expression, P is any statement and the notation $A\{E_1{\leftarrow}E_2\}$ denotes the array identical to A, except that the value at E_1 is E_2.

In order to reason about arrays, the following axioms, which define the meaning of the notation $A\{E_1{\leftarrow}E_2\}$, are needed.

The array axioms

$$\vdash \ A\{E_1{\leftarrow}E_2\}(E_1) \ = \ E_2$$

$$E_1 \neq E_3 \quad \Rightarrow \quad \vdash \ A\{E_1{\leftarrow}E_2\}(E_3) \ = \ A(E_3)$$

Example: We show

```
⊢ {A(X)=x ∧ A(Y)=y}
 BEGIN
   VAR R;
   R    := A(X);
   A(X) := A(Y);
   A(Y) := R
 END
  {A(X)=y ∧ A(Y)=x}
```

Working backwards using the array assignment axiom:

```
⊢ {A{Y←R}(X)=y ∧ A{Y←R}(Y)=x}
   A(Y) := R
  {A(X)=y ∧ A(Y)=x}
```

By precondition strengthening using $\vdash$ `A{Y←R}(Y) = R`

```
⊢ {A{Y←R}(X)=y ∧ R=x}
   A(Y) := R
  {A(X)=y ∧ A(Y)=x}
```

Continuing backwards

```
⊢ {A{X←A(Y)}{Y←R}(X)=y ∧ R=x}
   A(X) := A(Y)
  {A{Y←R}(X)=y ∧ R=x}
```

```
⊢ {A{X←A(Y)}{Y←A(X)}(X)=y ∧ A(X)=x}
   R := A(X)
  {A{X←A(Y)}{Y←R}(X)=y ∧ R=x}
```

Hence by the derived sequencing rule:

```
⊢ {A{X←A(Y)}{Y←A(X)}(X)=y ∧ A(X)=x}
   R := A(X); A(X) := A(Y); A(Y) := R
  {A(X)=y ∧ A(Y)=x}
```

By the array axioms (considering the cases `X=Y` and `X≠Y` separately), it follows that:

```
⊢ A{X←A(Y)}{Y←A(X)}(X)  =  A(Y)
```

Hence:

```
⊢ {A(Y)=y ∧ A(X)=x}
   R := A(X); A(X) := A(Y); A(Y) := R
  {A(X)=y ∧ A(Y)=x}
```

The desired result follows from the block rule.

□

Exercise 20

Show

```
⊢ {A(X)=x ∧ A(Y)=y ∧ X≠Y}
   A(X) := A(X) + A(Y);
   A(Y) := A(X) - A(Y);
   A(X) := A(X) - A(Y)
  {A(X)=y ∧ A(Y)=x}
```

Why is the precondition `X≠Y` necessary? □

Example: Suppose $\mathtt{C}_{sort}$ is a command that is intended to sort the first n elements of an array. To specify this formally, let $\mathtt{SORTED}(A, n)$ mean that:

$$A(1) \leq A(2) \leq \ldots \leq A(n)$$

A first attempt to specify that $\mathtt{C}_{sort}$ sorts is:

$$\{\mathtt{1} \leq \mathtt{N}\}\ \mathtt{C}_{sort}\ \{\mathtt{SORTED(A,N)}\}$$

This is not enough, however, because `SORTED(A,N)` can be achieved by simply zeroing the first `N` elements of `A`.

Exercise 21

Prove

```
⊢ {1≤N}
   FOR I:=1 UNTIL N DO A(I):=0
  {SORTED(A,N)}
```

□

It is necessary to require that the sorted array is a rearrangement, or permutation, of the original array.

To formalize this, let $\texttt{PERM}(A, A', N)$ mean that $A(1), A(2), \ldots, A(n)$ is a rearrangement of $A'(1), A'(2), \ldots, A'(n)$.

An improved specification that $\texttt{C}_{sort}$ sorts is then

$$\{\texttt{1}\leq\texttt{N} \wedge \texttt{A=a}\}\ \texttt{C}_{sort}\ \{\texttt{SORTED(A,N)} \wedge \texttt{PERM(A,a,N)}\}$$

However, this still is not correct[2].

Exercise 22

Prove

$$\begin{array}{l} \vdash \{\texttt{1}\leq\texttt{N} \wedge \texttt{A=a}\} \\ \quad \texttt{N:=1} \\ \quad \{\texttt{SORTED(A,N)} \wedge \texttt{PERM(A,a,N)}\} \end{array}$$

□

It is necessary to say explicitly that N is unchanged also. A correct specification is thus:

$$\{\texttt{1}\leq\texttt{N} \wedge \texttt{A=a} \wedge \texttt{N=n}\}\ \texttt{C}_{sort}\ \{\texttt{SORTED(A,N)} \wedge \texttt{PERM(A,a,N)} \wedge \texttt{N=n}\}$$

2.2 Soundness and completeness

It is clear from the discussion of the FOR-rule in Section 2.1.11 that it is not always straightforward to devise correct rules of inference. As discussed at the beginning of Chapter 2, it is very important that the axioms and rules be sound. There are two approaches to ensure this:

(i) Define the language by the axioms and rules of the logic.

(ii) Prove that the logic fits the language.

Approach (i) is called *axiomatic semantics*. The idea is to *define* the semantics of the language by requiring that it make the axioms and rules of inference true. It is then up to implementers to ensure that the logic matches the language. One snag with this approach is that most existing languages have already been defined in some other way (usually by informal and ambiguous natural language statements). An example of a language defined axiomatically is Euclid [29]. The other snag with axiomatic semantics is that it is known to be impossible to devise complete Floyd-Hoare logics for certain constructs (this is discussed further below). It could be argued that this is not a snag at all but an advantage, because it forces programming

[2] Thanks to an anonymous member of the 1990 class "Proving Programs Correct" for pointing this out to me.

languages to be made logically tractable. I have some sympathy for this latter view; it is clearly not the position taken by the designers of Ada.

Approach (ii) requires that the axioms and rules of the logic be proved valid. To do this, a mathematical model of states is constructed and then a function, $\mathsf{Meaning}$ say, is defined which takes an arbitrary command C to a function $\mathsf{Meaning}\ (C)$ from states to states. Thus $\mathsf{Meaning}\ (C)\ (s)$ denotes the state resulting from executing command C in state s. The specification $\{P\}C\{Q\}$ is then defined to be true if whenever P is true in a state s and $\mathsf{Meaning}\ (C)\ (s) = s'$ then Q is true in state s'. It is then possible to attempt to prove rigorously that all the axioms are true and that the rules of inference lead from true premisses to true conclusions. Actually carrying out this proof is likely to be quite tedious, especially if the programming language is at all complicated, and there are various technical details which require care (e.g. defining $\mathsf{Meaning}$ to correctly model non-termination). The precise formulation of such soundness proofs is not covered here, but details can be found in the text by Loeckx and Sieber [28].

Even if we are sure that our logic is sound, how can we be sure that every true specification can be proved? It might be the case that for some particular `P`, `Q` and `C` the specification $\{$`P`$\}$`C`$\{$`Q`$\}$ was true, but the rules of our logic were too weak to prove it (see Exercise 7 on page 25 for an example). A logic is said to be *complete* if every true statement in it is provable. There are various subtle technical problems in formulating precisely what it means for a Floyd-Hoare logic to be complete. For example, it is necessary to distinguish incompleteness arising due to incompleteness in the assertion language (e.g. arithmetic) from incompleteness due to inadequate axioms and rules for programming language constructs. The completeness of a Floyd-Hoare logic must thus be defined independently of that of its assertion language. Good introductions to this area can be found in Loeckx and Sieber [28] and Clarke's paper [9]. Clarke's paper also contains a discussion of his important results showing the impossibility of giving complete inference systems for certain combinations of programming language constructs. For example, he proves that it is impossible to give a sound and complete system for any language combining procedures as parameters of procedure calls, recursion, static scopes, global variables and internal procedures as parameters of procedure calls. These features are found in Algol 60, which thus cannot have a sound and complete Floyd-Hoare logic.

2.3 Some exercises

The exercises in this section have been taken from various sources, including Alagić and Arbib's book [1] and Cambridge University Tripos examinations.

Exercise 23

The exponentiation function *exp* satisfies:

$$\mathit{exp}(m, 0) = 1$$
$$\mathit{exp}(m, n{+}1) = m \times \mathit{exp}(m, n)$$

Devise a command C that uses repeated multiplication to achieve the following partial correctness specification:

$$\{X = x \ \wedge\ Y = y \ \wedge\ Y \geq 0\}\ C\ \{Z = \mathit{exp}(x, y) \ \wedge\ X = x \ \wedge\ Y = y\}$$

Prove that your command C meets this specification. □

Exercise 24

Show that:

```
⊢ {M≥0}
 BEGIN
   X:=0;
   FOR N:=1 UNTIL M DO X:=X+N
 END
  {X=(M×(M+1)) DIV 2}
```

Hint: Compare precondition with that of Exercise 19 on Page 34. □

Exercise 25

Deduce:

```
⊢ {S = (x×y)-(X×Y)}
 WHILE ¬ODD(X) DO
   BEGIN Y:=2×Y; X:=X DIV 2 END
  {S = (x×y)-(X×Y) ∧ ODD(X)}
```

□

Exercise 26

Deduce:

```
⊢ {S = (x×y)-(X×Y)}
 WHILE ¬(X=0) DO
   BEGIN
     WHILE ¬ODD(X) DO
       BEGIN Y:=2×Y; X:=X DIV 2 END;
     S:=S+Y;
     X:=X-1
   END
  {S = x×y}
```

□

Exercise 27

Deduce:

```
⊢ {X=x ∧ Y=y}
 BEGIN
   S:=0;
   WHILE ¬(X=0) DO
     BEGIN
       WHILE ¬ODD(X) DO
         BEGIN Y:=2×Y; X:=X DIV 2 END;
       S:=S+Y;
       X:=X-1
     END
 END
  {S = x×y}
```

□

Exercise 28

Prove the following invariant property.

```
⊢ {S = (x-X)×y ∧ Y=y}
 BEGIN
   VAR R;
   R:=0;
   WHILE ¬(R=Y) DO
     BEGIN S:=S+1; R:=R+1 END;
   X:=X-1
 END
  {S = (x-X)×y}
```

Hint: Show that `S = (x-X)×y + R` is an invariant for `S:=S+1; R:=R+1`. □

Exercise 29

Deduce:

```
⊢ {X=x ∧ Y=y}
 BEGIN
   S:=0;
   WHILE ¬(X=0) DO
     BEGIN
       VAR R;
       R:=0;
       WHILE ¬(R=Y) DO
         BEGIN S:=S+1; R:=R+1 END;
       X:=X-1
     END
 END
  {S = x×y}
```

□

Exercise 30

Using $\texttt{P}\times\texttt{X}^{\texttt{N}}\texttt{=}x^n$ as an invariant, deduce:

```
⊢ {X=x ∧ N=n}
  BEGIN
    P:=1;
    WHILE ¬(N=0) DO
      BEGIN
        IF ODD(N) THEN P:=P×X;
        N:=N DIV 2;
        X:=X×X
      END
  END
   {P = xⁿ}
```

□

Exercise 31

Prove that the command

```
BEGIN
  Z:=0;
  WHILE ¬(X=0) DO
    BEGIN
      IF ODD(X) THEN Z:=Z+Y;
      Y:=Y×2;
      X:=X DIV 2
    END
END
```

computes the product of the initial values of `X` and `Y` and leaves the result in `Z`. □

Exercise 32

Prove that the command

```
BEGIN
  Z:=1;
  WHILE N>0 DO
    BEGIN
      IF ODD(N) THEN Z:=Z×X;
      N:=N DIV 2;
      X:=X×X
    END
END
```

assigns x^n to `Z`, where x and n are the initial values of `X` and `N` respectively and we assume $n \geq 0$. □

Exercise 33

Devise a proof rule for a command

`REPEAT` *command* `UNTIL` *statement*

The meaning of `REPEAT C UNTIL S` is that `C` is executed and then `S` is tested; if the result is true, then nothing more is done, otherwise the whole `REPEAT` command is repeated. Thus `REPEAT C UNTIL S` is equivalent to `C; WHILE ¬S DO C`. □

Exercise 34

Use your REPEAT rule to deduce:

```
⊢ {S = C+R ∧ R<Y}
 REPEAT
   BEGIN S:=S+1; R:=R+1 END
 UNTIL R=Y
  {S = C+Y}
```

□

Exercise 35

Use your REPEAT rule to deduce:

```
⊢ {X=x ∧ Y=y}
 BEGIN
   S:=0;
   REPEAT
    BEGIN
     R:=0;
     REPEAT
       BEGIN S:=S+1; R:=R+1 END
     UNTIL R=Y;
     X:=X-1
    END
   UNTIL X=0
 END
  {S = x×y}
```

□

Exercise 36

Assume gcd(X,Y) satisfies:

```
⊢ (X>Y) ⇒ gcd(X,Y)=gcd(X-Y,Y)
⊢ gcd(X,Y)=gcd(Y,X)
⊢ gcd(X,X)=X
```

Prove:

```
⊢ {(A>0) ∧ (B>0) ∧ (gcd(A,B)=gcd(X,Y))}
 WHILE A>B DO A:=A-B;
 WHILE B>A DO B:=B-A
  {(0<B) ∧ (B≤A) ∧ (gcd(A,B)=gcd(X,Y))}
```

Hence, or otherwise, use your rule for REPEAT commands to prove:

```
⊢ {A=a ∧ B=b}
 REPEAT
  BEGIN
   WHILE A>B DO A:=A-B;
   WHILE B>A DO B:=B-A
  END
 UNTIL A=B
  {A=B ∧ A=gcd(a,b)}
```

□

Exercise 37

Prove:

```
⊢ {N≥1}
 BEGIN
   PROD=0;
   FOR X:=1 UNTIL N DO PROD := PROD+M
 END
  {PROD = M×N}
```

□

Exercise 38

Prove:

```
⊢ {X>0 ∧ Y>0}
 BEGIN
   S:=0;
   FOR I:=1 UNTIL X DO
     FOR J:=1 UNTIL Y DO
       S:=S+1
 END
  {S = X×Y}
```

□

Exercise 39

In some programming languages (e.g. ALGOLW) the following specification would be true:

```
⊢ {Y=3}
   FOR I:=1 UNTIL Y DO Y := Y+1
  {Y=6}
```

Explain why this cannot be deduced using the `FOR`-rule given on page 33. Design a more general `FOR`-rule, and use it to prove the above specification.

□

Exercise 40

(**Hard!**) Prove:

```
⊢ {1≤N ∧ A=a ∧ N=n}
   FOR I:=1 UNTIL N DO
     FOR J:=I+1 UNTIL N DO
       IF A(I)>A(J)
       THEN BEGIN
              VAR R;
              R    := A(I);
              A(I) := A(J);
              A(J) := R
            END
  {SORTED(A,N) ∧ PERM(A,a,N) ∧ N=n}
```

□

Outline solution

Warning: although I believe the outline proof below to be sound, I have not yet fully checked all the details and so there may be errors and omissions.

We outline a proof that:

```
⊢ {1≤N ∧ A=a ∧ N=n}
   FOR I:=1 UNTIL N DO
     FOR J:=I+1 UNTIL N DO
       IF A(I)>A(J)
       THEN BEGIN
             VAR R;
             R    := A(I);
             A(I) := A(J);
             A(J) := R
           END
  {SORTED(A,N) ∧ PERM(A,a,N) ∧ N=n}
```

Let `IFSWAP(A,I,J)` abbreviate the command:

```
       IF A(I)>A(J)
       THEN BEGIN
             VAR R;
             R    := A(I);
             A(I) := A(J);
             A(J) := R
            END
```

Then what has to be proved is:

```
⊢ {1≤N ∧ A=a ∧ N=n}
   FOR I:=1 UNTIL N DO
     FOR J:=I+1 UNTIL N DO
       IFSWAP(A,I,J)
  {SORTED(A,N) ∧ PERM(A,a,N) ∧ N=n}
```

By specification conjunction it is sufficient to prove.

```
⊢ {1≤N}
   FOR I:=1 UNTIL N DO
     FOR J:=I+1 UNTIL N DO
       IFSWAP(A,I,J)
  {SORTED(A,N)}
```

and

```
⊢ {1≤N ∧ A=a}
   FOR I:=1 UNTIL N DO
     FOR J:=I+1 UNTIL N DO
       IFSWAP(A,I,J)
  {PERM(A,a,N)}
```

and

```
⊢ {N=n}
   FOR I:=1 UNTIL N DO
     FOR J:=I+1 UNTIL N DO
       IFSWAP(A,I,J)
  {N=n}
```

The proof of the second and third of these are easy. For the second, it is necessary to use properties of the predicate PERM such as the following:

$\vdash$ `PERM(A,A,N)`

$\vdash$ `PERM(A,A',N)` $\wedge$ `1` $\leq$ `I` $\leq$ `N` $\wedge$ `1` $\leq$ `J` $\leq$ `N` $\Rightarrow$
   `PERM(A,A'{I←A'(J)}{J←A'(I)},N)`

Using these properties one can show that swapping two elements in an array preserves permutation, i.e.

```
⊢ {PERM(A,a,N) ∧ 1≤I ∧ I≤N}
       IFSWAP(A,I,J)
  {PERM(A,a,N)}
```

Using the FOR-rule twice on this specification, followed by precondition strengthening with PERM using the first property above, establishes that the sorting command only permutes the array. To show that it sorts the array into ascending order is more complicated.

Notice that each inner FOR loop results in the minimum element in A(I), ..., A(N) being placed at position I. To formulate this formally, define MIN(A,I,J) to mean that if 1 $\leq$ I $\leq$ J then A(I) is less than or equal to each of A(I+1), ..., A(J-1). It is intuitively clear that MIN has the properties:

```
⊢ MIN(A,I,J) ∧ A(I)≤A(J) ⇒ MIN(A,I,J+1)

⊢ MIN(A,I,J) ∧ A(I)≥A(J) ⇒  MIN(A{I←A(J)}{J←A(I)},I,J+1)
```

Using these properties it is straightforward to prove:

```
⊢ {MIN(A,I,J)} IFSWAP(A,I,J) {MIN(A,I,J+1)}
```

and hence by the FOR-rule

```
⊢ {1≤N ∧ MIN(A,I,I+1)}
     FOR J:=I+1 UNTIL N DO
       IFSWAP(A,I,J)
  {MIN(A,I,N)}
```

and hence, since $\vdash$ MIN(A,I,I+1), it follows by precondition strengthening that:

```
⊢ {1≤N}
     FOR J:=I+1 UNTIL N DO
       IFSWAP(A,I,J)
  {MIN(A,I,N)}
```

Using various properties of MIN and SORTED one can also show that

```
⊢ {SORTED(A,I-1) ∧ MIN(A,I-1,N+1) ∧ I≤J}
       IFSWAP(A,I,J)
⊢ {SORTED(A,I-1) ∧ MIN(A,I-1,N+1)}
```

and hence by the FOR-rule

```
⊢ {SORTED(A,I-1) ∧ MIN(A,I-1,N+1) ∧ I≤I+1}
     FOR J:=I+1 UNTIL N DO
       IFSWAP(A,I,J)
⊢ {SORTED(A,I-1) ∧ MIN(A,I-1,N+1)}
```

which simplifies, by precondition strengthening and postcondition weakening, to:

```
⊢ {SORTED(A,I-1) ∧ MIN(A,I-1,N+1)}
     FOR J:=I+1 UNTIL N DO
       IFSWAP(A,I,J)
⊢ {SORTED(A,I-1)}
```

hence by specification conjunction (using a previously proved result):

```
⊢ {SORTED(A,I-1) ∧ MIN(A,I-1,N+1) ∧ 1≤N}
     FOR J:=I+1 UNTIL N DO
       IFSWAP(A,I,J)
⊢ {SORTED(A,I-1) ∧ MIN(A,I,N)}
```

hence by postcondition weakening

```
⊢ {SORTED(A,I-1) ∧ MIN(A,I-1,N+1) ∧ 1≤N}
     FOR J:=I+1 UNTIL N DO
       IFSWAP(A,I,J)
⊢ {SORTED(A,I)}
```

hence by the `FOR`-rule (noticing that `SORTED(A,I)` is equivalent to `SORTED(A,(I+1)-1)`)

```
⊢ {SORTED(A,1-1) ∧ MIN(A,1-1,N+1) ∧ 1≤N}
   FOR I:=1 UNTIL N DO
     FOR J:=I+1 UNTIL N DO
       IFSWAP(A,I,J)
⊢ {SORTED(A,(N+1)-1)}
```

which simplifies to the desired result using `1-1=0` and properties of `SORTED` and `MIN`.

Chapter 3

Mechanizing Program Verification

The architecture of a simple program verifier is described. Its operation is justified with respect to the rules of Floyd-Hoare logic.

After doing only a few exercises, the following two things will be painfully clear:

(i) Proofs are typically long and boring (even if the program being verified is quite simple).

(ii) There are lots of fiddly little details to get right, many of which are trivial (e.g. proving $\vdash$ $(\mathtt{R}{=}\mathtt{X} \ \wedge \ \mathtt{Q}{=}\mathtt{0}) \ \Rightarrow \ (\mathtt{X} = \mathtt{R} + \mathtt{Y}{\times}\mathtt{Q})$).

Many attempts have been made (and are still being made) to automate proof of correctness by designing systems to do the boring and tricky bits of generating formal proofs in Floyd-Hoare logic. Unfortunately logicians have shown that it is impossible in principle to design a decision procedure to decide automatically the truth or falsehood of an arbitrary mathematical statement [13]. However, this does not mean that one cannot have procedures that will prove many useful theorems. The non-existence of a general decision procedure merely shows that one cannot hope to prove *everything* automatically. In practice, it is quite possible to build a system that will mechanize many of the boring and routine aspects of verification. This chapter describes one commonly taken approach to doing this.

Although it is impossible to decide automatically the truth or falsity of arbitrary statements, it *is* possible to check whether an arbitrary formal proof is valid. This consists in checking that the results occurring on each line of the proof are indeed either axioms or consequences of previous lines. Since proofs of correctness of programs are typically very long and boring, they often contain mistakes when generated manually. It is thus useful to check proofs mechanically, even if they can only be generated with human assistance.

3.1 Overview

In the previous chapter it was shown how to prove $\{P\}C\{Q\}$ by proving properties of the components of C and then putting these together (with the appropriate proof rule) to get the desired property of C itself. For example, to prove $\vdash \{P\}C_1;C_2\{Q\}$

first prove $\vdash \{P\}C_1\{R\}$ and $\vdash \{R\}C_2\{Q\}$ (for suitable R), and then deduce $\vdash \{P\}C_1;C_2\{Q\}$ by the sequencing rule.

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

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

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

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

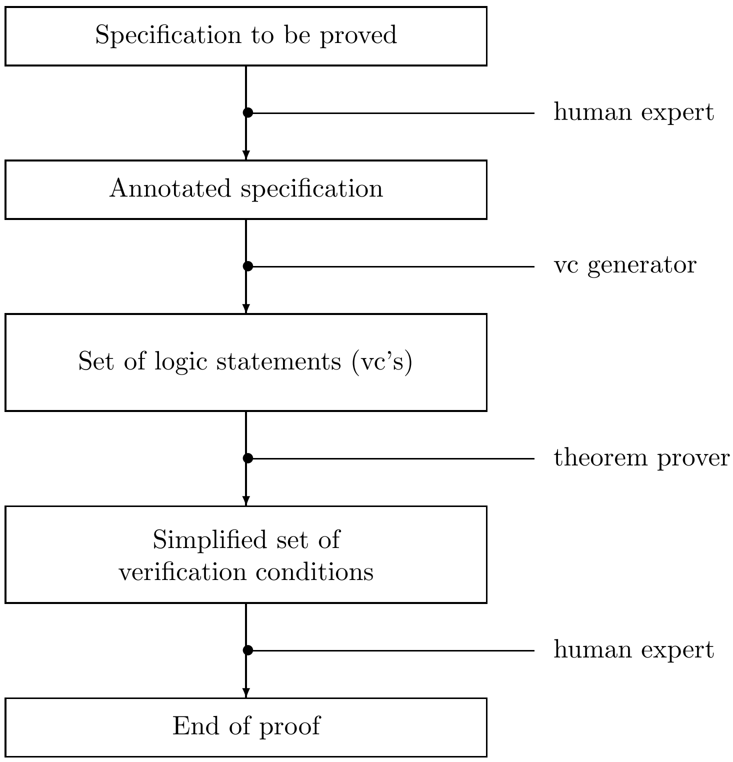
(because $\vdash \{\texttt{R=x} \wedge \texttt{X=y}\}\ \texttt{Y:=R}\ \{\texttt{Y=x} \wedge \texttt{X=y}\}$). By a similar argument this subgoal can be reduced to

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

which clearly follows from the assignment axiom.

This chapter describes how such a *goal oriented* method of proof can be formalized.

The verification system described here can be viewed as a proof checker that also provides some help with generating proofs. The following diagram gives an overview of the system.



The system takes as input a partial correctness specification annotated with mathematical statements describing relationships between variables. From the annotated specification the system generates a set of purely mathematical statements, called *verification conditions* (or vc's). In Section 3.5 it is shown that if these verification conditions are provable, then the original specification can be deduced from the axioms and rules of Floyd-Hoare logic.

The verification conditions are passed to a *theorem prover* program which attempts to prove them automatically; if it fails, advice is sought from the user. We will concentrate on those aspects pertaining to Floyd-Hoare logic and say very little about theorem proving here.

The aim of much current research is to build systems which reduce the role of the slow and expensive human expert to a minimum. This can be achieved by:

- reducing the number and complexity of the annotations required, and
- increasing the power of the theorem prover.

The next section explains how verification conditions work. In Section 3.5 their use is justified in terms of the axioms and rules of Floyd-Hoare logic. Besides being the basis for mechanical verification systems, verification conditions are a useful way of doing proofs by hand.

3.2 Verification conditions

The following sections describe how a goal oriented proof style can be formalized. To prove a goal $\{P\}C\{Q\}$, three things must be done. These will be explained in detail later, but here is a quick overview:

(i) The program C is *annotated* by inserting into it statements (often called *assertions*) expressing conditions that are meant to hold at various intermediate points. This step is tricky and needs intelligence and a good understanding of how the program works. Automating it is a problem of artificial intelligence.

(ii) A set of logic statements called *verification conditions* (vc's for short) is then generated from the annotated specification. This process is purely mechanical and easily done by a program.

(iii) The verification conditions are proved. Automating this is also a problem of artificial intelligence.

It will be shown that if one can prove all the verification conditions generated from $\{P\}C\{Q\}$ (where C is suitably annotated), then $\vdash \{P\}C\{Q\}$.

Since verification conditions are just mathematical statements, one can think of step 2 above as the 'compilation', or translation, of a verification problem into a conventional mathematical problem.

The following example will give a preliminary feel for the use of verification conditions.

Suppose the goal is to prove (see the example on page 28)

```
{T}
 BEGIN
   R:=X;
   Q:=0;
   WHILE Y≤R DO
     BEGIN R:=R-Y; Q:=Q+1 END
 END
{X = R+Y×Q ∧ R<Y}
```

This first step ((i) above) is to insert annotations. A suitable annotated specification is:

```
{T}
 BEGIN
   R:=X;
   Q:=0; {R=X ∧ Q=0} ⟵P1
   WHILE Y≤R DO {X = R+Y×Q} ⟵P2
     BEGIN R:=R-Y; Q:=Q+1 END
 END
{X = R+Y×Q ∧ R<Y}
```

The annotations P_1 and P_2 state conditions which are intended to hold *whenever* control reaches them. Control only reaches the point at which P_1 is placed once, but it reaches P_2 each time the `WHILE` body is executed and whenever this happens P_2 (i.e. `X=R+Y×Q`) holds, even though the values of `R` and `Q` vary. P_2 is an *invariant* of the `WHILE`-command.

The second step ((ii) above), which has yet to be explained, will generate the following four verification conditions:

(i) `T ⇒ (X=X ∧ 0=0)`

(ii) `(R=X ∧ Q=0) ⇒ (X = R+(Y×Q))`

(iii) `(X = R+(Y×Q)) ∧ Y≤R) ⇒ (X = (R-Y)+(Y×(Q+1)))`

(iv) `(X = R+(Y×Q)) ∧ ¬(Y≤R) ⇒  (X = R+(Y×Q) ∧ R<Y)`

Notice that these are statements of arithmetic; the constructs of our programming language have been 'compiled away'.

The third step ((iii) above) consists in proving these four verification conditions. These are all well within the capabilities of modern automatic theorem provers.

3.3 Annotation

An annotated command is a command with statements (called *assertions*) embedded within it. A command is said to be properly annotated if statements have been inserted at the following places:

(i) Before each command C_i (where $i > 1$) in a sequence C_1;C_2; … ;C_n which is *not* an assignment command,

(ii) After the word `DO` in `WHILE` and `FOR` commands.

Intuitively, the inserted assertions should express the conditions one expects to hold *whenever* control reaches the point at which the assertion occurs.

A properly annotated specification is a specification $\{P\}C\{Q\}$ where C is a properly annotated command.

Example: To be properly annotated, assertions should be at points ① and ② of the specification below:

```
{X=n}
 BEGIN
   Y:=1; ←—①
   WHILE X≠0 DO ←—②
     BEGIN Y:=Y×X; X:=X-1 END
 END
{X=0 ∧ Y=n!}
```

Suitable statements would be:

at ①: {Y = 1 ∧ X = n}
at ②: {Y×X! = n!}

□

The verification conditions generated from an annotated specification $\{P\}C\{Q\}$ are described by considering the various possibilities for C in turn. This process is justified in Section 3.5 by showing that $\vdash \{P\}C\{Q\}$ if all the verification conditions can be proved.

3.4 Verification condition generation

In this section a procedure is described for generating verification conditions for an annotated partial correctness specification $\{P\}C\{Q\}$. This procedure is *recursive* on C.

Assignment commands

The single verification condition generated by

$$\{P\}\ V\texttt{:=}E\ \{Q\}$$

is

$$P \ \Rightarrow\ Q\texttt{[}E/V\texttt{]}$$

The single verification condition generated by

$$\{P\}\ A\texttt{(}E_1\texttt{):=}E_2\ \{Q\}$$

is

$$P \Rightarrow Q\texttt{[}A\{E_1{\leftarrow}E_2\}/A\texttt{]}$$

Example: The verification condition for

$$\{\texttt{X=0}\}\ \texttt{X:=X+1}\ \{\texttt{X=1}\}$$

is

$$\texttt{X=0} \ \Rightarrow\ \texttt{(X+1)=1}$$

(which is clearly true). □

Verifications for array assignments are obtained by treating $A\texttt{(}E_1\texttt{):=}E_2$ as the ordinary assignment $A := A\{E_1{\leftarrow}E_2\}$ as discussed in Section 2.1.12.

One-armed conditional

The verification conditions generated by

$$\{P\}\ \texttt{IF}\ S\ \texttt{THEN}\ C\ \{Q\}$$

are

(i) $(P\ \wedge\ \neg S)\ \Rightarrow\ Q$

(ii) the verification conditions generated by

$$\{P\ \wedge\ S\}\ C\ \{Q\}$$

Example: The verification conditions for

$$\{\texttt{T}\}\ \texttt{IF X<0 THEN X:=-X}\ \{\texttt{X}\geq\texttt{0}\}$$

are $\texttt{T}\ \wedge\ \neg\texttt{(X<0)}\ \Rightarrow\ \texttt{X}\geq\texttt{0}$ together with the verification conditions for $\{\texttt{T}\ \wedge\ \texttt{(X<0)}\}\ \texttt{X:=-X}\ \{\texttt{X}\geq\texttt{0}\}$, i.e. $\texttt{T}\ \wedge\ \texttt{(X<0)}\ \Rightarrow\ \texttt{-X}\geq\texttt{0}$. The two vc's are thus:

(i) $\texttt{T} \wedge \neg\texttt{(X<0)} \Rightarrow \texttt{X}\geq\texttt{0}$

(ii) $\texttt{T} \wedge \texttt{(X<0)} \Rightarrow \texttt{-X}\geq\texttt{0}$

These are equivalent to $\texttt{X}\geq\texttt{0} \Rightarrow \texttt{X}\geq\texttt{0}$ and $\texttt{X<0} \Rightarrow \texttt{-X}\geq\texttt{0}$, respectively, which are both clearly true. □

Two-armed conditional

The verification conditions generated from

$$\{P\}\ \texttt{IF}\ S\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2\ \{Q\}$$

are

(i) the verification conditions generated by

$$\{P\ \wedge\ S\}\ C_1\ \{Q\}$$

(ii) the verification conditions generated by

$$\{P\ \wedge\ \neg S\}\ C_2\ \{Q\}$$

Exercise 41
What are the verification conditions for the following specification?

$$\{\texttt{T}\}\ \texttt{IF X}\geq\texttt{Y THEN MAX:=X ELSE MAX:=Y}\ \{\texttt{MAX=max(X,Y)}\}$$

Do they follow from the assumptions about `max(X,Y)` given in the example on page 26? □

If $C_1;\ldots;C_n$ is properly annotated, then (see page 50) it must be of one of the two forms:

1. $C_1;\ \ldots\ ;C_{n-1};\{R\}C_n$, or

2. $C_1;\ \ldots\ ;C_{n-1};V := E$.

where, in both cases, $C_1;\ \ldots\ ;C_{n-1}$ is a properly annotated command.

Sequences

1. The verification conditions generated by

$$\{P\}\ C_1\texttt{;}\ \ldots\ \texttt{;}C_{n-1}\texttt{;}\ \{R\}\ C_n\ \{Q\}$$

(where C_n is not an assignment) are:

   (a) the verification conditions generated by

   $$\{P\}\ C_1\texttt{;}\ \ldots\ \texttt{;}C_{n-1}\ \{R\}$$

   (b) the verification conditions generated by

   $$\{R\}\ C_n\ \{Q\}$$

2. The verification conditions generated by

$$\{P\}\ C_1\texttt{;}\ \ldots\ \texttt{;}C_{n-1}\texttt{;}V\texttt{:=}E\ \{Q\}$$

are the verification conditions generated by

$$\{P\}\ C_1\texttt{;}\ \ldots\ \texttt{;}C_{n-1}\ \{Q\texttt{[}E/V\texttt{]}\}$$

Example: The verification conditions generated from

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

are those generated by

$$\{\texttt{X=x}\ \wedge\ \texttt{Y=y}\}\ \texttt{R:=X; X:=Y}\ \{\texttt{(X=y}\ \wedge\ \texttt{Y=x)[R/Y]}\}$$

which, after doing the substitution, simplifies to

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

The verification conditions generated by this are those generated by

$$\{\texttt{X=x}\ \wedge\ \texttt{Y=y}\}\ \texttt{R:=X}\ \{\texttt{(X=y}\ \wedge\ \texttt{R=x)[Y/X]}\}$$

which, after doing the substitution, simplifies to

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

The only verification condition generated by this is

$$\texttt{X=x}\ \wedge\ \texttt{Y=y}\ \Rightarrow\ \texttt{(Y=y}\ \wedge\ \texttt{R=x)[X/R]}$$

which, after doing the substitution, simplifies to

$$\texttt{X=x}\ \wedge\ \texttt{Y=y}\ \Rightarrow\ \texttt{Y=y}\ \wedge\ \texttt{X=x}$$

which is obviously true. □

The procedure for generating verification conditions from blocks involves checking the syntactic condition that the local variables of the block do not occur in the precondition or postcondition. The need for this is clear from the side condition in the block rule (see page 24); this will be explained in more detail when the procedure for generating verification conditions is justified in Section 3.5.

Blocks

The verification conditions generated by

$$\{P\}\ \texttt{BEGIN VAR}\ V_1\texttt{;}\ \ldots\ \texttt{;VAR}\ V_n\texttt{;}C\ \texttt{END}\ \{Q\}$$

are

(i) the verification conditions generated by $\{P\}C\{Q\}$, and

(ii) the syntactic condition that none of V_1, ..., V_n occur in either P or Q.

Example: The verification conditions for

`{X=x ∧ Y=y} BEGIN VAR R; R:=X; X:=Y; Y:=R END {X=y ∧ Y=x}`

are those generated by `{X=x ∧ Y=y} R:=X; X:=Y; Y:=R {X=y ∧ Y=x}` (since R does not occur in `{X=x ∧ Y=y}` or `{X=y ∧ Y=x}`). See the previous example for the verification conditions generated by this. □

Exercise 42
What are the verification conditions for the following specification?

`{X = R+(Y×Q)} BEGIN R:=R-Y; Q:=Q+1 END {X = R+(Y×Q)}`

□

Exercise 43
What are the verification conditions for the following specification?

`{X=x} BEGIN VAR X; X:=1 END {X=x}`

Relate your answer to this exercise to your answer to Exercise 7 on page 25. □

A correctly annotated specification of a `WHILE`-command has the form

$$\{P\}\ \texttt{WHILE}\ S\ \texttt{DO}\ \{R\}\ C\ \{Q\}$$

Following the usage on page 28, the annotation R is called an invariant.

WHILE-commands

The verification conditions generated from

$$\{P\}\ \texttt{WHILE}\ S\ \texttt{DO}\ \{R\}\ C\ \{Q\}$$

are

(i) $P \ \Rightarrow\ R$

(ii) $R\ \wedge\ \neg S\ \Rightarrow\ Q$

(iii) the verification conditions generated by $\{R\ \wedge\ S\}\ C\{R\}$.

Example: The verification conditions for

$$\begin{array}{l}\{\texttt{R=X}\ \wedge\ \texttt{Q=0}\}\\ \ \texttt{WHILE Y}\leq\texttt{R DO}\ \{\texttt{X=R+Y}\times\texttt{Q}\}\\ \quad\texttt{BEGIN R:=R-Y; Q=Q+1 END}\\ \{\texttt{X = R+(Y}\times\texttt{Q)}\ \wedge\ \texttt{R<Y}\}\end{array}$$

are:

(i) $\texttt{R=X}\ \wedge\ \texttt{Q=0}\ \Rightarrow\ \texttt{(X = R+(Y}\times\texttt{Q))}$

(ii) $\texttt{X = R+Y}\times\texttt{Q}\ \wedge\ \neg\texttt{(Y}\leq\texttt{R)}\ \Rightarrow\ \texttt{(X = R+(Y}\times\texttt{Q)}\ \wedge\ \texttt{R<Y)}$

together with the verification condition for

$$\begin{array}{l}\{\texttt{X = R+(Y}\times\texttt{Q)}\ \wedge\ \texttt{(Y}\leq\texttt{R)}\}\\ \ \texttt{BEGIN R:=R-Y; Q:=Q+1 END}\\ \{\texttt{X=R+(Y}\times\texttt{Q)}\}\end{array}$$

which (see Exercise 42) consists of the single condition

(iii) $\texttt{X = R+(Y}\times\texttt{Q)}\ \wedge\ \texttt{(Y}\leq\texttt{R)}\ \Rightarrow\ \texttt{X = (R-Y)+(Y}\times\texttt{(Q+1))}$

The WHILE-command specification is thus true if (i), (ii) and (iii) hold, i.e.

$$\begin{array}{l}\vdash\ \{\texttt{R=X}\ \wedge\ \texttt{Q=0}\}\\ \ \ \texttt{WHILE Y}\leq\texttt{R DO}\\ \quad\ \texttt{BEGIN R:=R-Y; Q:=Q+1 END}\\ \quad\ \{\texttt{X = R+(Y}\times\texttt{Q)}\ \wedge\ \texttt{R<Y}\}\end{array}$$

if

$$\vdash\ \texttt{R=X}\ \wedge\ \texttt{Q=0}\ \Rightarrow\ \texttt{(X = R+(Y}\times\texttt{Q))}$$

and

$$\vdash\ \texttt{X = R+(Y}\times\texttt{Q)}\ \wedge\ \neg\texttt{(Y}\leq\texttt{R)}\ \Rightarrow\ \texttt{(X = R+(Y}\times\texttt{Q)}\ \wedge\ \texttt{R<Y)}$$

and

$$\vdash\ \texttt{X = R+(Y}\times\texttt{Q)}\ \wedge\ \texttt{(Y}\leq\texttt{R)}\ \Rightarrow\ \texttt{X = (R-Y)+(Y}\times\texttt{(Q+1))}$$

□

Exercise 44

What are the verification conditions generated by the annotated program for computing `n!` (the factorial of `n`) given in the example on page 51? □

A correctly annotated specification of a `FOR`-command has the form

$$\{P\}\ \mathtt{FOR}\ V\mathtt{:=}E_1\ \mathtt{UNTIL}\ E_2\ \mathtt{DO}\ \{R\}\ C\ \{Q\}$$

`FOR`-commands

The verification conditions generated from

$$\{P\}\ \mathtt{FOR}\ V\mathtt{:=}E_1\ \mathtt{UNTIL}\ E_2\ \mathtt{DO}\ \{R\}\ C\ \{Q\}$$

are

(i) $P \;\Rightarrow\; R\mathtt{[}E_1/V\mathtt{]}$

(ii) $R\mathtt{[}E_2{+}1/V\mathtt{]} \;\Rightarrow\; Q$

(iii) $P \;\wedge\; E_2 < E_1 \;\Rightarrow\; Q$

(iv) the verification conditions generated by

$$\{R \;\wedge\; E_1 \leq\; V \;\wedge\; V \leq E_2\}\ C\ \{R\mathtt{[}V+1/V\mathtt{]}\}$$

(v) the syntactic condition that neither V, nor any variable occurring in E_1 or E_2, is assigned to inside C.

Example: The verification conditions generated by

$$\{\mathtt{X=0}\ \wedge\ \mathtt{1}\leq\mathtt{M}\}$$
$$\mathtt{FOR\ N:=1\ UNTIL\ M\ DO}\ \{\mathtt{X=((N-1)}\times\mathtt{N)\ DIV\ 2}\}\ \mathtt{X:=X+N}$$
$$\{\mathtt{X\ =\ (M}\times\mathtt{(M+1))\ DIV\ 2}\}$$

are

(i) $\mathtt{X=0}\ \wedge\ \mathtt{1}\leq\mathtt{M}\ \Rightarrow\ \mathtt{X=((1-1)}\times\mathtt{1)\ DIV\ 2}$

(ii) $\mathtt{X\ =\ (((M+1)-1)}\times\mathtt{(M+1))\ DIV\ 2}\ \Rightarrow\ \ \mathtt{X\ =\ (M}\times\mathtt{(M+1))\ DIV\ 2}$

(iii) $\mathtt{X=0}\ \wedge\ \mathtt{1}\leq\mathtt{M}\ \wedge\ \mathtt{M<1}\ \Rightarrow\ \ \mathtt{X\ =\ (M}\times\mathtt{(M+1))\ DIV\ 2}$

(iv) The verification condition generated by

$$\{\mathtt{X\ =\ ((N-1)}\times\mathtt{N)\ DIV\ 2}\ \wedge\ \mathtt{1}\leq\mathtt{N}\ \wedge\ \mathtt{N}\leq\mathtt{M}\}$$
$$\mathtt{X:=X+N}$$
$$\{\mathtt{X\ =\ (((N+1)-1)}\times\mathtt{(N+1))\ DIV\ 2}\}$$

which, after some simplification, is

$$\mathtt{X} = ((\mathtt{N-1}) \times \mathtt{N})\ \mathtt{DIV\ 2}\ \wedge\ \mathtt{1} \leq \mathtt{N}\ \wedge\ \mathtt{N} \leq \mathtt{M}$$
$$\Rightarrow$$
$$\mathtt{X+N} = (\mathtt{N} \times (\mathtt{N+1}))\ \mathtt{DIV\ 2}$$

which is true since

$$\frac{(\mathtt{N}-\mathtt{1})\times\mathtt{N}}{\mathtt{2}}+\mathtt{N}=\frac{\mathtt{2N}+(\mathtt{N}-\mathtt{1})\times\mathtt{N}}{\mathtt{2}}$$

$$=\frac{\mathtt{2N}+\mathtt{N}^2-\mathtt{N}}{\mathtt{2}}$$

$$=\frac{\mathtt{N}+\mathtt{N}^2}{\mathtt{2}}$$

$$=\frac{\mathtt{N}\times(\mathtt{N}+\mathtt{1})}{\mathtt{2}}$$

(Exercise: justify this calculation in the light of the fact that

$$(x+y)\ \mathtt{DIV}\ z \neq (x\ \mathtt{DIV}\ z)+(y\ \mathtt{DIV}\ z)$$

as is easily seen by taking x, y and z to be `3`, `5` and `8`, respectively.)

(v) Neither `N` or `M` is assigned to in `X:=X+N`

□

3.5 Justification of verification conditions

It will be shown in this section that an annotated specification $\{P\}C\{Q\}$ is provable in Floyd-Hoare logic (i.e. $\vdash\{P\}C\{Q\}$) if the verification conditions generated by it are provable. This shows that the verification conditions are *sufficient*, but not that they are necessary. In fact, the verification conditions are the weakest sufficient conditions, but we will neither make this more precise nor go into details here. An in-depth study of preconditions can be found in Dijkstra's book [10].

It is easy to show (see the exercise below) that the verification conditions are not necessary, i.e. that the verification conditions for $\{P\}C\{Q\}$ not being provable doesn't imply that $\vdash\{P\}C\{Q\}$ cannot be deduced.

Exercise 45
Show that

(i) The verification conditions from the annotated specification

`{T} WHILE F DO {F} X:=0 {T}`

are not provable.

(ii) $\vdash$ `{T} WHILE F DO X:=0 {T}`

□

The argument that the verification conditions are sufficient will be by *induction* on the structure of C. Such inductive arguments have two parts. First, it is shown that the result holds for assignment commands. Second, it is shown that when C is not an assignment command, then if the result holds for the constituent commands of C (this is called the *induction hypothesis*), then it holds also for C. The first of these parts is called the *basis* of the induction and the second is called the *step*. From the basis and the step it follows that the result holds for all commands.

Assignments

The only verification condition for $\{P\}V{:=}E\{Q\}$ is $P \Rightarrow Q\texttt{[}E/V\texttt{]}$. If this is provable, then as $\vdash\ \{Q\texttt{[}E/V\texttt{]}\}V{:=}E\{Q\}$ (by the assignment axiom on page 19) it follows by precondition strengthening (page 21) that $\vdash\ \{P\}V := E\{Q\}$.

One-armed conditionals

If the verification conditions for $\{P\}$ $\texttt{IF}$ S $\texttt{THEN}$ C $\{Q\}$ are provable, then $\vdash$ $P \wedge \neg S \ \Rightarrow\ Q$ and all the verification conditions for $\{P \wedge S\}$ C $\{Q\}$ are provable. Hence by the induction hypothesis $\vdash\ \{P \wedge S\}$ C $\{Q\}$ and hence by the one-armed conditional rule (page 26) it follows that $\vdash\ \{P\}$ $\texttt{IF}$ S $\texttt{THEN}$ C $\{Q\}$.

Two-armed conditionals

If the verification conditions for $\{P\}$ $\texttt{IF}$ S $\texttt{THEN}$ C_1 $\texttt{ELSE}$ C_2 $\{Q\}$ are provable, then the verification conditions for both $\{P\ \wedge\ S\}$ C_1 $\{Q\}$ and $\{P\ \wedge\ \neg S\}$ C_2 $\{Q\}$ are provable. By the induction hypothesis we can assume that $\vdash\ \{P\ \wedge\ S\}$ C_1 $\{Q\}$ and $\vdash\ \{P\ \wedge\ \neg S\}$ C_2 $\{Q\}$. Hence by the two-armed conditional rule (page 26) $\vdash\ \{P\}$ $\texttt{IF}$ S $\texttt{THEN}$ C_1 $\texttt{ELSE}$ C_2 $\{Q\}$.

Sequences

There are two cases to consider:

(i) If the verification conditions for $\{P\}$ $C_1\texttt{;}\ldots\texttt{;}C_{n-1}\texttt{;}\{R\}C_n$ $\{Q\}$ are provable, then the verification conditions for $\{P\}$ $C_1\texttt{;}\ldots\texttt{;}C_{n-1}$ $\{R\}$ and $\{R\}$ C_n $\{Q\}$ must both be provable and hence by induction we have $\vdash$ $\{P\}$ $C_1\texttt{;}\ldots\texttt{;}C_{n-1}$ $\{R\}$ and $\vdash\ \{R\}$ C_n $\{Q\}$. Hence by the sequencing rule (page 23) $\vdash\ \{P\}$ $C_1\texttt{;}\ldots\texttt{;}\ C_{n-1}\texttt{;}C_n$ $\{Q\}$.

(ii) If the verification conditions for $\{P\}$ $C_1\texttt{;}\ldots\texttt{;}C_{n-1}\texttt{;}V\ :=\ E$ $\{Q\}$ are provable, then it must be the case that the verification conditions for $\{P\}$ $C_1\texttt{;}\ldots\texttt{;}C_{n-1}$ $\{Q\texttt{[}E/V\texttt{]}\}$ are also provable and hence by induction we have $\vdash\ \{P\}$ $C_1\texttt{;}\ldots\texttt{;}C_{n-1}$ $\{Q\texttt{[}E/V\texttt{]}\}$. It then follows by the assignment axiom that $\vdash\ \{Q\texttt{[}E/V\texttt{]}\}$ $V\ :=\ E$ $\{Q\}$, hence by the sequencing rule $\vdash\ \{P\}$ $C_1\texttt{;}\ldots\texttt{;}C_{n-1}\texttt{;}V := E\{Q\}$.

Blocks

If the verification conditions for $\{P\}$`BEGIN VAR` V_1`;` $\ldots$ `;VAR` V_n`;`C `END` $\{Q\}$ are provable, then the verification conditions for $\{P\}\ C\ \{Q\}$ are provable and V_1, $\ldots$, V_n do not occur in P or Q. By induction $\vdash \{P\}\ C\ \{Q\}$ hence by the block rule (page 24) $\vdash\ \{P\}$ `BEGIN VAR` V_1`;` $\ldots$ `;VAR` V_n`;`C `END` $\{Q\}$.

`WHILE`-commands

If the verification conditions for $\{P\}$ `WHILE` S `DO` $\{R\}\ C\ \{Q\}$ are provable, then $\vdash\ P \Rightarrow R$, $\vdash\ (R\ \wedge\ \neg S)\ \Rightarrow\ Q$ and the verification conditions for $\{R\ \wedge\ S\}\ C\ \{R\}$ are provable. By induction $\vdash\ \{R\ \wedge\ S\}\ C\ \{R\}$, hence by the `WHILE`-rule (page 28) $\vdash\ \{R\}$ `WHILE` S `DO` C $\{R\ \wedge\ \neg S\}$, hence by the consequence rules (see page 22) $\vdash\ \{P\}$ `WHILE` S `DO` C $\{Q\}$.

`FOR`-commands

Finally, if the verification conditions for

$$\{P\}\ \texttt{FOR}\ V := E_1\ \texttt{UNTIL}\ E_2\ \texttt{DO}\ \{R\}\ C\ \{Q\}$$

are provable, then

(i) $\vdash\ P\ \Rightarrow\ R\texttt{[}E_1/V\texttt{]}$

(ii) $\vdash\ R\texttt{[}E_2+1/V\texttt{]}\ \Rightarrow\ Q$

(iii) $\vdash\ P\ \wedge\ E_2 < E_1\ \Rightarrow\ Q$

(iv) The verification conditions for

$$\{R\ \wedge\ E_1 \leq V\ \wedge\ V \leq E_2\}\ C\ \{R\texttt{[}V+1/V\texttt{]}\}$$

are provable.

(v) Neither V, nor any variable in E_1 or E_2, is assigned to in C.

By induction $\vdash\ \{R \wedge E_1 \leq V\ \wedge V \leq E_2\}\ C\ \{R\texttt{[}V+1/V\texttt{]}\}$, hence by the `FOR`-rule

$$\vdash\ \{R\texttt{[}E_1/V\texttt{]}\ \wedge\ E_1 \leq E_2\}\ \texttt{FOR}\ V := E_1\ \texttt{UNTIL}\ E_2\ \texttt{DO}\ C\ \{R\texttt{[}E_2+1/V\texttt{]}\}$$

hence by (i), (ii) and the consequence rules

(vi) $\vdash\ \{P\ \wedge\ E_1 \leq E_2\}$ `FOR` $V := E_1$ `UNTIL` E_2 `DO` C $\{Q\}$.

Now by the `FOR`-axiom (see page 32) with P instantiated to $P\ \wedge\ E_2 < E_1$, followed by Precondition Strengthening (to eliminate the repeated $E_2 < E_1$):

$$\vdash\ \{P\ \wedge\ E_2 < E_1\}\ \texttt{FOR}\ V := E_1\ \texttt{UNTIL}\ E_2\ \texttt{DO}\ C\ \{P\ \wedge\ E_2 < E_1\},$$

hence by the consequence rules and (iii)

$$\vdash \{P \ \wedge \ E_2 < E_1\} \ \texttt{FOR} \ V := E_1 \ \texttt{UNTIL} \ E_2 \ \texttt{DO} \ C \ \{Q\}.$$

Combining this last specification with (vi) using specification disjunction (page 22) yields

$$\vdash \{P \ \wedge \ E_2 < E_1) \ \vee \ (P \ \wedge \ E_1 \leq E_2)\} \ \texttt{FOR} \ V := E_1 \ \texttt{UNTIL} \ E_2 \ \texttt{DO} \ C \ \{Q \ \vee \ Q\}$$

Now $\vdash \ Q \ \vee \ Q \ \Rightarrow \ Q$ and

$$\vdash \ (P \ \wedge \ E_2 < E_1) \ \vee \ (P \ \wedge \ E_1 \leq E_2) \ \Leftrightarrow \ P \ \wedge \ (E_2 < E_1 \ \vee \ E_1 \leq E_2)$$

but $\vdash \ E_2 < E_1 \ \vee \ E_1 \leq E_2$, hence

$$\vdash \ P \ \Rightarrow \ (P \ \wedge E_2 < E_1) \ \vee \ (P \ \wedge \ E_1 \leq E_2)$$

and so one can conclude:

$$\vdash \ \{P\} \ \texttt{FOR} \ V := E_1 \ \texttt{UNTIL} \ E_2 \ \texttt{DO} \ C \ \{Q\}$$

Thus the verification conditions for the `FOR`-command are sufficient.

Exercise 46
Annotate the specifications in Exercises 24 to 32 (they start on page 39) and then generate the corresponding verification conditions. □

Exercise 47
Devise verification conditions for commands of the form

$$\texttt{REPEAT} \ C \ \texttt{UNTIL} \ S$$

(See Exercise 33, page 41.) □

Exercise 48
Do Exercises 34–38 using verification conditions. □

Exercise 49
With the rules given, can one prove by induction on the structure of C that if no variable occurring in P is assigned to in C, then $\vdash \ \{P\} \ C\{P\}$? □

Exercise 50
Consider the following alternative scheme, due to Silas Brown, for generating VCs from annotated `WHILE`-commands.

WHILE-commands

Alternative verification conditions generated from

$$\{P\}\ \texttt{WHILE}\ S\ \texttt{DO}\ \{R\}\ C\ \{Q\}$$

are

(i) $P \ \wedge \ S \ \ \Rightarrow \ R$

(ii) $P \ \wedge \ \neg S \ \Rightarrow \ Q$

(iii) the verification conditions generated by $\{R\}\ C\{(Q \ \wedge \ \neg S) \ \vee \ (R \ \wedge \ S)\}$.

Either justify these VCs, or find a counterexample. □

Chapter 4

Total Correctness

The axioms and rules of Floyd-Hoare logic are extended to total correctness. Verification conditions for total correctness specifications are given.

In Section 1.3 the notation $[P]\ C\ [Q]$ was introduced for the total correctness specification that C halts in a state satisfying Q whenever it is executed in a state satisfying P. At the end of the section describing the `WHILE`-rule (Section 2.1.10), it is shown that the rule is not valid for total correctness specifications. This is because `WHILE`-commands may introduce non-termination. None of the other commands can introduce non-termination, and thus the rules of Floyd-Hoare logic can be used.

4.1 Axioms and rules for non-looping commands

Replacing curly brackets by square ones results in the following axioms and rules.

Assignment axiom for total correctness

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

Precondition strengthening for total correctness

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

Postcondition weakening for total correctness

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

Specification conjunction for total correctness

$$\frac{\vdash\ [P_1]\ C\ [Q_1], \qquad \vdash\ [P_2]\ C\ [Q_2]}{\vdash\ [P_1 \wedge P_2]\ C\ [Q_1 \wedge Q_2]}$$

Specification disjunction for total correctness

$$\frac{\vdash\ [P_1]\ C\ [Q_1], \qquad \vdash\ [P_2]\ C\ [Q_2]}{\vdash\ [P_1 \vee P_2]\ C\ [Q_1 \vee Q_2]}$$

Sequencing rule for total correctness

$$\frac{\vdash\ [P]\ C_1\ [Q], \qquad \vdash\ [Q]\ C_2\ [R]}{\vdash\ [P]\ C_1 ; C_2\ [R]}$$

Derived sequencing rule for total correctness

$$\frac{\begin{array}{ll} & \vdash\ P \Rightarrow P_1 \\ \vdash\ [P_1]\ C_1\ [Q_1] & \vdash\ Q_1 \Rightarrow P_2 \\ \vdash\ [P_2]\ C_2\ [Q_2] & \vdash\ Q_2 \Rightarrow P_3 \\ . & . \\ . & . \\ . & . \\ \vdash\ [P_n]\ C_n\ [Q_n] & \vdash\ Q_n \Rightarrow Q \end{array}}{\vdash\ [P]\ C_1;\ \ldots\ ;\ C_n\ [Q]}$$

Block rule for total correctness

$$\frac{\vdash\ [P]\ C\ [Q]}{\vdash\ [P]\ \texttt{BEGIN VAR}\ V_1;\ \ldots;\ \texttt{VAR}\ V_n;\ C\ \texttt{END}\ [Q]}$$

Where none of the variables $V_1, \ldots, V_n$ occur in P or Q.

Derived block rule for total correctness

$$\frac{\begin{array}{ll} & \vdash\ P \Rightarrow P_1 \\ \vdash\ [P_1]\ C_1\ [Q_1] & \vdash\ Q_1 \Rightarrow P_2 \\ \vdash\ [P_2]\ C_2\ [Q_2] & \vdash\ Q_2 \Rightarrow P_3 \\ . & . \\ . & . \\ . & . \\ \vdash\ [P_n]\ C_n\ [Q_n] & \vdash\ Q_n \Rightarrow Q \end{array}}{\vdash\ [P]\ \texttt{BEGIN VAR}\ V_1;\ \ldots\ \texttt{VAR}\ V_n ; C_1;\ \ldots\ ;\ C_n\ [Q]}$$

Where none of the variables $V_1, \ldots, V_n$ occur in P or Q.

Conditional rules for total correctness

$$\frac{\vdash\ [P \wedge S]\ C\ [Q], \qquad \vdash\ P \wedge \neg S \Rightarrow Q}{\vdash\ [P]\ \texttt{IF}\ S\ \texttt{THEN}\ C\ [Q]}$$

$$\frac{\vdash\ [P \wedge S]\ C_1\ [Q], \qquad \vdash\ [P \wedge \neg S]\ C_2\ [Q]}{\vdash\ [P]\ \texttt{IF}\ S\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2\ [Q]}$$

`FOR`-axiom and rule for total correctness

$$\frac{\vdash [P \wedge \texttt{(}E_1 \leq V\texttt{)} \wedge \texttt{(}V \leq E_2\texttt{)}]\ C\ [P\texttt{[}V\texttt{+1/}V\texttt{]}]}{\vdash [P\texttt{[}E_1/V\texttt{]}\wedge\texttt{(}E_1 \leq E_2\texttt{)}]\ \texttt{FOR}\ V := E_1\ \texttt{UNTIL}\ E_2\ \texttt{DO}\ C\ [P\texttt{[}E_2\texttt{+1/}V\texttt{]}]}$$

Where neither V, nor any variable occurring in E_1 or E_2, is assigned to in the command C.

$$\vdash\ [P \wedge (E_2 < E_1)]\ \texttt{FOR}\ V := E_1\ \texttt{UNTIL}\ E_2\ \texttt{DO}\ C\ [P]$$

4.2 Derived rule for total correctness of non-looping commands

The rules just given are formally identical to the corresponding rules of Floyd-Hoare logic, except that they have [and] instead of { and }. It is thus clear that the following is a valid derived rule.

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

If C contains no `WHILE`-commands.

4.3 The termination of assignments

Note that the assignment axiom for total correctness states that assignment commands always terminate, which implicitly assumes that all function applications in expressions terminate. This might not be the case if functions could be defined recursively. For example, consider the assignment: X `:=` $fact(-1)$, where $fact(n)$ is defined recursively by:

$$fact(n)\ =\ \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times fact(n-1)$$

It is also assumed that erroneous expressions like $1/0$ do not cause problems. Most programming languages will cause an error stop when division by zero is encountered. However, in our logic it follows that:

$$\vdash\ [\texttt{T}]\ \texttt{X :=}\ 1/0\ [\texttt{X} = 1/0]$$

i.e. the assignment `X :=` $1/0$ always halts in a state in which the condition $\texttt{X} = 1/0$ holds. This assumes that $1/0$ denotes some value that `X` can have. There are two possibilities:

(i) $1/0$ denotes some number;

(ii) $1/0$ denotes some kind of 'error value'.

It seems at first sight that adopting (ii) is the most natural choice. However, this makes it tricky to see what arithmetical laws should hold. For example, is $(1/0) \times 0$ equal to 0 or to some 'error value'? If the latter, then it is no longer the case that $n \times 0 = 0$ is a valid general law of arithmetic? It is possible to make everything work with undefined and/or error values, but the resultant theory is a bit messy. We shall assume here that arithmetic expressions always denote numbers, but in some cases exactly what the number is will be not fully specified. For example, we will assume that m/n denotes a number for any m and n, but the only property of "/" that will be assumed is:

$$\neg(n = 0) \quad\Rightarrow\quad (m/n) \times n = m$$

It is not possible to deduce anything about $m/0$ from this, in particular it is not possible to deduce $(m/0) \times 0 = 0$. In fact, sometimes it may be assumed that numbers are integers. In this case, the defining property of "/" is:

$$\neg(n = 0) \;\wedge\; p \times n = m \quad\Rightarrow\quad (m/n) = p$$

4.4 `WHILE`-rule for total correctness

`WHILE`-commands are the only commands in our little language that can cause non-termination, they are thus the only kind of command with a non-trivial termination rule. The idea behind the `WHILE`-rule for total correctness is that to prove `WHILE` S `DO` C terminates one must show that some non-negative quantity decreases on each iteration of C. This decreasing quantity is called a *variant*. In the rule below, the variant is E, and the fact that it decreases is specified with an auxiliary variable n. An extra hypothesis, $\vdash\ P \wedge S \Rightarrow E \geq 0$, ensures the variant is non-negative.

`WHILE`-rule for total correctness

$$\frac{\vdash\ [P \wedge S \wedge (E = n)]\ C\ [P \wedge (E < n)], \qquad \vdash\ P \wedge S \Rightarrow E \geq 0}{\vdash\ [P]\ \texttt{WHILE}\ S\ \texttt{DO}\ C\ [P \wedge \neg S]}$$

where E is an integer-valued expression and n is an auxiliary variable not occurring in P, C, S or E.

Example: We show:

$$\vdash\ [\texttt{Y} > 0]\ \texttt{WHILE Y}\leq\texttt{R DO BEGIN R:=R-Y; Q:=Q+1 END}\ [\texttt{T}]$$

Take

$$\begin{array}{lcl} P & = & \texttt{Y} > \texttt{0} \\ S & = & \texttt{Y} \leq \texttt{R} \\ E & = & \texttt{R} \\ C & = & \texttt{BEGIN R:=R-Y Q:=Q+1 END} \end{array}$$

We want to show $\vdash\ [P]$ `WHILE` S `DO` C $[\texttt{T}]$. By the `WHILE`-rule for total correctness it is sufficiant to show:

(i) $\vdash\ [P \wedge S \wedge (E = \mathtt{n})]\ C\ [P \wedge (E < \mathtt{n})]$

(ii) $\vdash\ P \wedge S \Rightarrow E \geq 0$

The first of these, (i), can be proved by establishing

$$\vdash\ \{P \wedge S \wedge (E = \mathtt{n})\}\ C\ \{P \wedge (E < \mathtt{n})\}$$

and then using the total correctness rule for non-looping commands. The verification condition for this partial correctness specification is:

$$\mathtt{Y} > 0\ \wedge\ \mathtt{Y} \leq \mathtt{R}\ \wedge\ \mathtt{R} = \mathtt{n}\quad\Rightarrow\quad(\mathtt{Y} > 0\ \wedge\ \mathtt{R} < \mathtt{n})[\mathtt{Q}{+}1/\mathtt{Q}][\mathtt{R}{-}\mathtt{Y}/\mathtt{R}]$$

i.e.

$$\mathtt{Y} > 0\ \wedge\ \mathtt{Y} \leq \mathtt{R}\ \wedge\ \mathtt{R} = \mathtt{n}\quad\Rightarrow\quad\mathtt{Y} > 0\ \wedge\ \mathtt{R}{-}\mathtt{Y} < \mathtt{n}$$

which follows from the laws of arithmetic.

The second subgoal, (ii), is just $\vdash\ \mathtt{Y} > 0\ \wedge\ \mathtt{Y} \leq \mathtt{R} \Rightarrow \mathtt{R} \geq 0$, which follows from the laws of arithmetic. □

4.5 Termination specifications

As already discussed in Section 1.3, the relation between partial and total correctness is informally given by the equation:

Total correctness = Termination + Partial correctness.

This informal equation above can now be represented by the following two formal rule of inferences.

$$\frac{\vdash\ \{P\}\ C\ \{Q\},\qquad \vdash\ [P]\ C\ [\mathtt{T}]}{\vdash\ [P]\ C\ [Q]}$$

$$\frac{\vdash\ [P]\ C\ [Q]}{\vdash\ \{P\}\ C\ \{Q\},\qquad \vdash\ [P]\ C\ [\mathtt{T}]}$$

Exercise 51

Think about whether these two rules are derivable from the other rules? □

4.6 Verification conditions for termination

The idea of verification conditions is easily extended to deal with total correctness.

To generate verification conditions for `WHILE`-commands, it is necessary to add a variant as an annotation in addition to an invariant. No other extra annotations are needed for total correctness. We assume this is added directly after the invariant, surrounded by square brackets. A correctly annotated total correctness specification of a `WHILE`-command thus has the form

$$[P]\ \texttt{WHILE}\ S\ \texttt{DO}\ \{R\}[E]\ C\ [Q]$$

where R is the invariant and E the variant. Note that the variant is intended to be a non-negative expression that decreases each time around the `WHILE` loop. The other annotations, which are enclosed in curly brackets, are meant to be conditions that are true whenever control reaches them. The use of square brackets around variant annotations is meant to be suggestive of this difference.

The rules for generating verification conditions from total correctness specifications are now given in the same format as the rules for generating partial correctness verification conditions given in Section 3.4.

4.7 Verification condition generation

Assignment commands

The single verification condition generated by

$$[P]\ V\texttt{:=}E\ [Q]$$

is

$$P \ \Rightarrow\ Q\texttt{[}E/V\texttt{]}$$

Example: The verification condition for

$$[\texttt{X=0}]\ \texttt{X:=X+1}\ [\texttt{X=1}]$$

is

$$\texttt{X=0}\ \Rightarrow\ \texttt{(X+1)=1}$$

This is the same as for partial correctness. □

One-armed conditional

The verification conditions generated by

$$[P]\ \texttt{IF}\ S\ \texttt{THEN}\ C\ [Q]$$

are

(i) $(P\ \wedge\ \neg S)\ \Rightarrow\ Q$

(ii) the verifications generated by $[P\ \wedge\ S]\ C\ [Q]$

Example: The verification conditions for

$$[\texttt{T}]\ \texttt{IF X<0 THEN X:=-X}\ [\texttt{X}\geq\texttt{0}]$$

are $\texttt{T} \wedge \neg\texttt{(X<0)} \Rightarrow \texttt{X}\geq\texttt{0}$ together with the verification conditions for $[\texttt{T} \wedge \texttt{(X<0)}]\ \texttt{X:=-X}\ [\texttt{X}\geq\texttt{0}]$, i.e. $\texttt{T} \wedge \texttt{(X<0)} \Rightarrow \texttt{-X}\geq\texttt{0}$. The two verification conditions are thus: $\texttt{T} \wedge \neg\texttt{(X>0)} \Rightarrow \texttt{X}\geq\texttt{0}$ and $\texttt{T} \wedge \texttt{(X<0)} \Rightarrow \texttt{-X}\geq\texttt{0}$. These are both clearly true. □

Two-armed conditional

The verification conditions generated from

$$[P]\ \texttt{IF}\ S\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2\ [Q]$$

are

(i) the verification conditions generated by $[P\ \wedge\ S]\ C_1\ [Q]$

(ii) the verifications generated by $[P\ \wedge\ \neg S]\ C_2\ [Q]$

Exercise 52
What are the verification conditions for the following specification?

$$[\texttt{T}]\ \texttt{IF X}\geq\texttt{Y THEN MAX:=X ELSE MAX:=Y}\ [\texttt{MAX=max(X,Y)}]$$

□

If $C_1;\ldots;C_n$ is properly annotated, then (see page 50) it must be of one of the two forms:

1. $C_1;\ \ldots\ ;C_{n-1};\{R\}C_n$, or

2. $C_1;\ \ldots\ ;C_{n-1};V := E$.

where, in both cases, $C_1;\ \ldots\ ;C_{n-1}$ is a properly annotated command.

Sequences

1. The verification conditions generated by

$$[P]\ C_1\,;\,\ldots\,;C_{n-1}\,;\ \{R\}\ C_n\ [Q]$$

(where C_n is not an assignment) are:

(a) the verification conditions generated by

$$[P]\ C_1\,;\,\ldots\,;C_{n-1}\ [R]$$

(b) the verification conditions generated by

$$[R]\ C_n\ [Q]$$

2. The verification conditions generated by

$$[P]\ C_1\,;\,\ldots\,;C_{n-1}\,;V\,\texttt{:=}E\ [Q]$$

are the verification conditions generated by

$$[P]\ C_1\,;\,\ldots\,;C_{n-1}\ [Q\texttt{[}E/V\texttt{]}]$$

Example: The verification conditions generated from

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

are those generated by

$$[\texttt{X=x} \wedge \texttt{Y=y}]\ \texttt{R:=X; X:=Y}\ [\texttt{(X=y} \wedge \texttt{Y=x)[R/Y]}]$$

which, after doing the substitution, simplifies to

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

The verification conditions generated by this are those generated by

$$[\texttt{X=x} \wedge \texttt{Y=y}]\ \texttt{R:=X}\ [\texttt{(X=y} \wedge \texttt{R=x)[Y/X]}]$$

which, after doing the substitution, simplifies to

$$[\texttt{X=x} \wedge \texttt{Y=y}]\ \texttt{R:=X}\ [\texttt{Y=y} \wedge \texttt{R=x}].$$

The only verification condition generated by this is

$$\texttt{X=x} \wedge \texttt{Y=y} \Rightarrow \texttt{(Y=y} \wedge \texttt{R=x)[X/R]}$$

which, after doing the substitution, simplifies to

$$\texttt{X=x} \wedge \texttt{Y=y} \Rightarrow \texttt{Y=y} \wedge \texttt{X=x}$$

which is obviously true. □

Blocks

The verification conditions generated by

$$[P]\ \texttt{BEGIN VAR}\ V_1\texttt{;}\ \ldots\ \texttt{;VAR}\ V_n\texttt{;}C\ \texttt{END}\ [Q]$$

are

(i) the verification conditions generated by $[P]\ C\ [Q]$, and

(ii) the syntactic condition that none of V_1, ..., V_n occur in either P or Q.

Example: The verification conditions for

```
[X=x ∧ Y=y] BEGIN VAR R; R:=X; X:=Y; Y:=R END [X=y ∧ Y=x]
```

are those generated by `[X=x ∧ Y=y] R:=X; X:=Y; Y:=R [X=y ∧ Y=x]` (since R does not occur in `[X=x ∧ Y=y]` or `[X=y ∧ Y=x]`). See the previous example for the verification conditions generated by this. □

A correctly annotated specification of a `WHILE`-command has the form

$$[P]\ \texttt{WHILE}\ S\ \texttt{DO}\ \{R\}[E]\ C\ [Q]$$

`WHILE`-commands

The verification conditions generated from

$$[P]\ \texttt{WHILE}\ S\ \texttt{DO}\ \{R\}[E]\ C\ [Q]$$

are

(i) $P \ \Rightarrow\ R$

(ii) $R\ \wedge\ \neg S\ \Rightarrow\ Q$

(iii) $R\ \wedge\ S\ \Rightarrow\ E \geq 0$

(iv) the verification conditions generated by

$$[R\ \wedge\ S\ \wedge\ (E = n)]\ C[R\ \wedge\ (E < n)]$$

where n is an auxiliary variable not occurring in P, C, S R, E or Q.

Example: The verification conditions for

```
[R=X ∧ Q=0]
 WHILE Y≤R DO {X=R+Y×Q}[R]
   BEGIN R:=R-Y; Q=Q+1 END
 [X = R+(Y×Q) ∧ R<Y]
```

are:

(i) `R=X` $\wedge$ `Q=0` $\Rightarrow$ `(X = R+(Y`$\times$`Q))`

(ii) `X = R+Y`$\times$`Q` $\wedge$ $\neg$`(Y`$\leq$`R)` $\Rightarrow$ `(X = R+(Y`$\times$`Q)` $\wedge$ `R<Y)`

(iii) `X = R+Y`$\times$`Q` $\wedge$ `Y`$\leq$`R` $\Rightarrow$ `R`$\geq$`0`

together with the verification condition for

$$\begin{array}{l}[\texttt{X = R+(Y}\times\texttt{Q)} \wedge \texttt{(Y}\leq\texttt{R)} \wedge \texttt{(R=n)}]\\ \;\texttt{BEGIN R:=R-Y; Q:=Q+1 END}\\ [\texttt{X=R+(Y}\times\texttt{Q)} \wedge \texttt{(R<n)}]\end{array}$$

which (exercise for the reader) consists of the single condition

(iv)

$$\texttt{X = R+(Y}\times\texttt{Q)} \wedge \texttt{(Y}\leq\texttt{R)} \wedge \texttt{(R=n)} \Rightarrow \texttt{X = (R-Y)+(Y}\times\texttt{(Q+1))} \wedge \texttt{((R-Y)<n)}$$

But this isn't true (take `Y=0`)! $\Box$

Exercise 53

Explain why one would not expect to be able to prove the verification conditions of the last example. Change the specification and annotations so that provable verifications are generated. $\Box$

A correctly annotated specification of a `FOR`-command has the form

$$\{P\}\ \texttt{FOR}\ V\texttt{:=}E_1\ \texttt{UNTIL}\ E_2\ \texttt{DO}\ \{R\}\ C\ \{Q\}$$

`FOR`-commands

The verification conditions generated from

$$[P]\ \texttt{FOR}\ V\texttt{:=}E_1\ \texttt{UNTIL}\ E_2\ \texttt{DO}\ \{R\}\ C\ [Q]$$

are

(i) $P \;\Rightarrow\; R\texttt{[}E_1/V\texttt{]}$

(ii) $R\texttt{[}E_2{+}1/V\texttt{]} \;\Rightarrow\; Q$

(iii) $P \;\wedge\; E_2 < E_1 \;\Rightarrow\; Q$

(iv) the verification conditions generated by

$$[R \;\wedge\; E_1 \leq\; V \;\wedge\; V \leq E_2]\ C\ [R\texttt{[}V+1/V\texttt{]}]$$

(v) the syntactic condition that neither V, nor any variable occurring in E_1 or E_2, is assigned to inside C.

Example: The verification conditions generated by

```
[X=0 ∧ 1≤M]
 FOR N:=1 UNTIL M DO {X=((N-1)×N) DIV 2} X:=X+N
[X = (M×(M+1)) DIV 2]
```

are

(i) `X=0 ∧ 1≤M ⇒ X=((1-1)×1) DIV 2`

(ii) `X = (((M+1)-1)×(M+1)) DIV 2 ⇒  X = (M×(M+1)) DIV 2`

(iii) `X=0 ∧ 1≤M ∧ M<1 ⇒  X = (M×(M+1)) DIV 2`

(iv) The verification condition generated by

```
[X = ((N-1)×N) DIV 2 ∧ 1≤N ∧ N≤M]
 X:=X+N
[X = (((N+1)-1)×(N+1)) DIV 2]
```

(v) The syntactic condition that neither `N` or `M` is assigned to in `X:=X+N`.

These verification conditionss are proved in the example on page 57. □

We leave it as an exercise for the reader to extend the argument given in Section 3.5 to a justification of the total correcness verification conditions.

Chapter 5

Program Refinement

Floyd-Hoare Logic is a method of proving that existing programs meet their specifications. It can also be used as a basis for 'refining' specifications to programs – i.e. as the basis for a programming methodology.

5.1 Introduction

The task of a programmer can be viewed as taking a specification consisting of a precondition P and postcondition Q and then coming up with a command C such that $\vdash\ [P]\ C\ [Q]$.

Theories of refinement present rules for 'calculating' programs C from specification P and Q. A key idea, due to Ralph Back [3] of Finland (and subsequently rediscovered by both Joseph Morris [35] and Carroll Morgan [34]), is to introduce a new class of programming constructs, called specifications. These play the same syntactic role as commands, but are not directly executable though they are guaranteed to achieve a given postcondition from a given precondition. The resulting generalized programming language contains pure specifications, pure code and mixtures of the two. Such languages are called *wide spectrum* languages.

The approach taken here[1] follows the style of refinement developed by Morgan, but is founded on Floyd-Hoare logic, rather than on Dijkstra's theory of weakest preconditions (see Section 7.6.2). This foundation is a bit more concrete and syntactical than the traditional one: a specification is identified with its set of possible implementations and refinement is represented as manipulations on sets of ordinary commands. This approach aims to convey the 'look and feel' of (Morgan style) refinement using the notational and conceptual ingredients introduced in the preceding chapters.

The notation $[P,\ Q]$ will be used for specifications, and thus:

$$[P,\ Q]\ =\ \{\ C\ \mid\ \ \vdash\ [P]\ C\ [Q]\ \}$$

The process of refinement will then consist of a sequence of steps that make systematic design decisions to narrow down the sets of possible implementations until

[1] The approach to refinement described here is due to Paul Curzon. Mark Staples and Joakim Von Wright provided some feedback on an early draft, which I have incorporated

a unique implementation is reached. Thus a refinement of a specification $\mathcal{S}$ to an implementation C has the form:

$$\mathcal{S} \supseteq \mathcal{S}_1 \supseteq \mathcal{S}_2 \cdots \supseteq \mathcal{S}_n \supseteq \{C\}$$

The initial specification $\mathcal{S}$ has the form $[P,\ Q]$ and each intermediate specification $\mathcal{S}_i$ is obtained from its predecessor $\mathcal{S}_{i-1}$ by the application of a *refinement law*.

In the literature $\mathcal{S} \supseteq \mathcal{S}'$ is normally written $\mathcal{S} \sqsubseteq \mathcal{S}'$. The use of "$\supseteq$" here, instead of the more abstract "$\sqsubseteq$", reflects the concrete interpretation of refinement as the narrowing down of sets of implementations.

5.2 Refinement laws

The refinement laws are derived from the axioms and rules of Floyd-Hoare Logic. In order to state these laws, the usual notation for commands is extended to sets of commands as follows ($\mathcal{C}$, $\mathcal{C}_1$, $\mathcal{C}_2$ etc. range over *sets* of commands):

$$\begin{array}{ll}
\mathcal{C}_1\texttt{;}\ \cdots\ \texttt{;}\mathcal{C}_n & = \{\ C_1\texttt{;}\ \cdots\ \texttt{;}C_n \ \mid\ C_1 \in \mathcal{C}_1\ \wedge\ \cdots\ \wedge\ C_n \in \mathcal{C}_n\ \} \\
\texttt{BEGIN VAR } V_1\texttt{;}\ \cdots\ \texttt{VAR } V_n\texttt{;}\ \mathcal{C}\ \texttt{END} & = \{\ \texttt{BEGIN VAR } V_1\texttt{;}\ \cdots\ \texttt{VAR } V_n\texttt{;}\ C\ \texttt{END}\ \mid\ C \in \mathcal{C}\ \} \\
\texttt{IF } S \texttt{ THEN } \mathcal{C} & = \{\ \texttt{IF } S \texttt{ THEN } C\ \mid\ C \in \mathcal{C}\ \} \\
\texttt{IF } S \texttt{ THEN } \mathcal{C}_1 \texttt{ ELSE } \mathcal{C}_2 & = \{\ \texttt{IF } S \texttt{ THEN } C_1 \texttt{ ELSE } C_2\ \mid\ C_1 \in \mathcal{C}_1\ \wedge\ C_2 \in \mathcal{C}_2\ \} \\
\texttt{WHILE } S \texttt{ DO } \mathcal{C} & = \{\ \texttt{WHILE } S \texttt{ DO } C\ \mid\ C \in \mathcal{C}\ \}
\end{array}$$

This notation for sets of commands can be viewed as constituting a wide spectrum language.

Note that such sets of commands are *monotonic* with respect to refinement (i.e. inclusion). If $\mathcal{C} \supseteq \mathcal{C}'$, $\mathcal{C}_1 \supseteq \mathcal{C}_1'$, ... , $\mathcal{C}_n \supseteq \mathcal{C}_n'$ then:

$$\begin{array}{ll}
\mathcal{C}_1\texttt{;}\ \cdots\ \texttt{;}\mathcal{C}_n & \supseteq \mathcal{C}_1'\texttt{;}\ \cdots\ \texttt{;}\mathcal{C}_n' \\
\texttt{BEGIN VAR } V_1\texttt{;}\ \cdots\ \texttt{VAR } V_n\texttt{;}\ \mathcal{C}\ \texttt{END} & \supseteq \texttt{BEGIN VAR } V_1\texttt{;}\ \cdots\ \texttt{VAR } V_n\texttt{;}\ \mathcal{C}'\ \texttt{END} \\
\texttt{IF } S \texttt{ THEN } \mathcal{C} & \supseteq \texttt{IF } S \texttt{ THEN } \mathcal{C}' \\
\texttt{IF } S \texttt{ THEN } \mathcal{C}_1 \texttt{ ELSE } \mathcal{C}_2 & \supseteq \texttt{IF } S \texttt{ THEN } \mathcal{C}_1' \texttt{ ELSE } \mathcal{C}_2' \\
\texttt{WHILE } S \texttt{ DO } \mathcal{C} & \supseteq \texttt{WHILE } S \texttt{ DO } \mathcal{C}'
\end{array}$$

This monotonicity shows that a command can be refined by separately refining its constituents.

The following 'laws' follow directly from the definitions above and the axioms and rules of Floyd-Hoare logic.

The Skip Law

$$[P,\ P]\ \supseteq\ \{\texttt{SKIP}\}$$

Derivation

$$\begin{array}{lll} C \in \{\texttt{SKIP}\} & & \\ \quad\Leftrightarrow\ \ C\ =\ \texttt{SKIP} & & \\ \quad\Rightarrow\ \ \vdash [P]\ C\ [P] & & \text{(Skip Axiom)} \\ \quad\Leftrightarrow\ \ C \in [P,\ P] & & \text{(Definition of } [P,\ P]) \end{array}$$

The Assignment Law

$$[P\texttt{[}E/V\texttt{]},\ P]\ \supseteq\ \{V\ \texttt{:=}\ E\}$$

Derivation

$$\begin{array}{lll} C \in \{V\ \texttt{:=}\ E\} & & \\ \quad\Leftrightarrow\ \ C\ =\ V\ \texttt{:=}\ E & & \\ \quad\Rightarrow\ \ \vdash [P\texttt{[}E/V\texttt{]}]\ C\ [P] & & \text{(Assignment Axiom)} \\ \quad\Leftrightarrow\ \ C \in [P\texttt{[}E/V\texttt{]},\ P] & & \text{(Definition of } [P\texttt{[}E/V\texttt{]},\ P]) \end{array}$$

Derived Assignment Law

$$[P,\ Q]\ \supseteq\ \{V\texttt{:=}E\}$$

$$\text{provided}\ \vdash\ P\ \Rightarrow\ Q\texttt{[}E/V\texttt{]}$$

Derivation

$$\begin{array}{lll} C \in \{V\ \texttt{:=}\ E\} & & \\ \quad\Leftrightarrow\ \ C\ =\ V\ \texttt{:=}\ E & & \\ \quad\Rightarrow\ \ \vdash [Q\texttt{[}E/V\texttt{]}]\ C\ [Q] & & \text{(Assignment Axiom)} \\ \quad\Rightarrow\ \ \vdash [P]\ C\ [Q] & & \text{(Precondition Strengthening \& } \vdash\ P \Rightarrow Q\texttt{[}E/V\texttt{]}) \\ \quad\Leftrightarrow\ \ C \in [P,\ Q] & & \text{(Definition of } [P,\ Q]) \end{array}$$

Precondition Weakening

$$[P,\ Q]\ \supseteq\ [R,\ Q]$$

$$\text{provided}\ \vdash\ \ P\ \Rightarrow\ R$$

Derivation

$$\begin{array}{lll} C \in [R,\ Q] & & \\ \quad\Leftrightarrow\ \ \vdash [R]\ C\ [Q] & & \text{(Definition of } [R,\ Q]) \\ \quad\Rightarrow\ \ \vdash [P]\ C\ [Q] & & \text{(Precondition Strengthening \& } \vdash\ P \Rightarrow R) \\ \quad\Leftrightarrow\ \ C \in [P,\ Q] & & \text{(Definition of } [P,\ Q]) \end{array}$$

Postcondition Strengthening

$$[P,\ Q]\ \supseteq\ [P,\ R]$$

$$\text{provided}\ \vdash\ \ R\ \Rightarrow\ Q$$

Derivation

$$\begin{array}{lll}
C \in [P,\ R] & & \\
\quad \Leftrightarrow & \vdash\ [P]\ C\ [R] & \text{(Definition of } [R,\ Q]\text{)} \\
\quad \Rightarrow & \vdash\ [P]\ C\ [Q] & \text{(Postcondition Weakening \& } \vdash\ R \Rightarrow Q\text{)} \\
\quad \Leftrightarrow & C \in [P,\ Q] & \text{(Definition of } [P,\ Q]\text{)}
\end{array}$$

The Sequencing Law

$$[P,\ Q]\ \supseteq\ [P,\ R]\ ;\ [R,\ Q]$$

Derivation

$$\begin{array}{lll}
C \in [P,\ R]\ ;\ [R,\ Q] & & \\
\quad \Leftrightarrow & C \in \{\ C_1\ ;\ C_2\ \mid\ C_1 \in [P,\ R]\ \&\ C_2 \in [R,\ Q]\} & \text{(Definition of } \mathcal{C}_1\ ;\ \mathcal{C}_2\text{)} \\
\quad \Leftrightarrow & C \in \{\ C_1\ ;\ C_2\ \mid\ \vdash\ [P]\ C_1\ [R]\ \&\ \vdash\ [R]\ C_2\ [Q]\} & \text{(Definition of } [P,\ R] \text{ and } [R,\ Q]\text{)} \\
\quad \Rightarrow & C \in \{\ C_1\ ;\ C_2\ \mid\ \vdash\ [P]\ C_1\ ;\ C_2\ [Q]\} & \text{(Sequencing Rule)} \\
\quad \Rightarrow & \vdash\ [P]\ C\ [Q] & \\
\quad \Leftrightarrow & C \in [P,\ Q] & \text{(Definition of } [P,\ Q]\text{)}
\end{array}$$

The Block Law

$$[P,\ Q]\ \supseteq\ \texttt{BEGIN VAR}\ V\texttt{;}\ [P,\ Q]\ \texttt{END}$$

where V does not occur in P or Q

Derivation

$$\begin{array}{lll}
C \in \texttt{BEGIN VAR}\ V\ ;\ [P,\ Q]\ \texttt{END} & & \\
\quad \Leftrightarrow & C \in \{\texttt{BEGIN VAR}\ V\ ;\ C'\ \texttt{END}\ \mid & \\
 & \qquad C' \in [P,\ Q]\} & \text{(Definition of } \texttt{BEGIN VAR}\ V\ ;\ \mathcal{C}\ \texttt{END}\text{)} \\
\quad \Leftrightarrow & C \in \{\texttt{BEGIN VAR}\ V\ ;\ C'\ \texttt{END}\ \mid & \\
 & \qquad \vdash\ [P]\ C'\ [Q]\} & \text{(Definition of } [P,\ Q]\text{)} \\
\quad \Rightarrow & C \in \{\texttt{BEGIN VAR}\ V\ ;\ C'\ \texttt{END}\ \mid & \\
 & \qquad \vdash\ [P]\ \texttt{BEGIN VAR}\ V\ ;\ C'\ \texttt{END}\ [Q]\} & \text{(Block Rule \& } V \text{ not in } P \text{ or } Q\text{)} \\
\quad \Rightarrow & \vdash\ [P]\ C\ [Q] & \\
\quad \Leftrightarrow & C \in [P,\ Q] & \text{(Definition of } [P,\ Q]\text{)}
\end{array}$$

The One-armed Conditional Law

$$[P,\ Q]\ \supseteq\ \texttt{IF}\ S\ \texttt{THEN}\ [P \wedge S,\ Q]$$

$$\text{provided}\ \vdash\ \ P \wedge \neg S\ \Rightarrow\ Q$$

Derivation

$$\begin{array}{lll}
\multicolumn{2}{l}{C \in \texttt{IF}\ S\ \texttt{THEN}\ [P \wedge S,\ Q]} & \\
\Leftrightarrow & C \in \{\texttt{IF}\ S\ \texttt{THEN}\ C' \mid & \\
 & \quad C' \in [P \wedge S,\ Q]\} & (\text{Definition of } \texttt{IF}\ S\ \texttt{THEN}\ \mathcal{C}) \\
\Leftrightarrow & C \in \{\texttt{IF}\ S\ \texttt{THEN}\ C' \mid & \\
 & \quad \vdash\ [P \wedge S]\ C'\ [Q]\} & (\text{Definition of } [P \wedge S,\ Q]) \\
\Rightarrow & C \in \{\texttt{IF}\ S\ \texttt{THEN}\ C' \mid & \\
 & \quad \vdash\ [P]\ \texttt{IF}\ S\ \texttt{THEN}\ C'\ [Q]\} & (\text{One-armed Conditional Rule \& } \vdash\ P \wedge \neg S \Rightarrow Q) \\
\Rightarrow & \vdash\ [P]\ C\ [Q] & \\
\Leftrightarrow & C \in [P,\ Q] & (\text{Definition of } [P,\ Q])
\end{array}$$

The Two-armed Conditional Law

$$[P,\ Q]\ \supseteq\ \texttt{IF}\ S\ \texttt{THEN}\ [P \wedge S,\ Q]\ \texttt{ELSE}\ [P \wedge \neg S,\ Q]$$

Derivation

$$\begin{array}{lll}
\multicolumn{2}{l}{C \in \texttt{IF}\ S\ \texttt{THEN}\ [P{\wedge}S,\ Q]\ \texttt{ELSE}\ [P{\wedge}\neg S,\ Q]} & \\
\Leftrightarrow & C \in \{\texttt{IF}\ S\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2 \mid & \\
 & \quad C_1 \in [P{\wedge}S,\ Q]\ \&\ C_2 \in [P{\wedge}\neg S,\ Q]\} & (\text{Definition of } \texttt{IF}\ S\ \texttt{THEN}\ \mathcal{C}_1\ \texttt{ELSE}\ \mathcal{C}_2) \\
\Leftrightarrow & C \in \{\texttt{IF}\ S\ \texttt{THEN}\ C_1\ \texttt{THEN}\ C_2 \mid & \\
 & \quad \vdash\ [P{\wedge}S]\ C_1\ [Q]\ \&\ \vdash\ [P{\wedge}\neg S]\ C_2\ [Q]\} & (\text{Definition of } [P{\wedge}S,\ Q]\ \&\ [P{\wedge}\neg S,\ Q]) \\
\Rightarrow & C \in \{\texttt{IF}\ S\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2 \mid & \\
 & \quad \vdash\ [P]\ \texttt{IF}\ S\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2\ [Q]\} & (\text{Two-armed Conditional Rule}) \\
\Rightarrow & \vdash\ [P]\ C\ [Q] & \\
\Leftrightarrow & C \in [P,\ Q] & (\text{Definition of } [P,\ Q])
\end{array}$$

The While Law

$$[P,\ P \wedge \neg S]\ \supseteq\ \texttt{WHILE}\ S\ \texttt{DO}\ [P \wedge S \wedge (E\texttt{=}n),\ P \wedge (E\texttt{<}n)]$$

$$\text{provided}\ \vdash\ \ P \wedge S \Rightarrow\ E \geq 0$$

where E is an integer-valued expression and n is an identifier not occurring in P, S or E.

Derivation

$$\begin{array}{lll}
\multicolumn{2}{l}{C \in \texttt{WHILE}\ S\ \texttt{DO}\ [P \wedge S \wedge (E = n),\ P \wedge (E < n)]} & \\
\Leftrightarrow & C \in \{\texttt{WHILE}\ S\ \texttt{DO}\ C' \mid & \\
 & \quad C' \in [P \wedge S \wedge (E = n),\ P \wedge (E < n)]\} & (\text{Definition of } \texttt{WHILE}\ S\ \texttt{DO}\ \mathcal{C}) \\
\Leftrightarrow & C \in \{\texttt{WHILE}\ S\ \texttt{DO}\ C' \mid & (\text{Definition of} \\
 & \quad \vdash\ [P \wedge S \wedge (E = n)]\ C'\ [P \wedge (E < n)]\} & \quad [P \wedge S \wedge (E = n),\ P \wedge (E < n)]) \\
\Rightarrow & C \in \{\texttt{WHILE}\ S\ \texttt{DO}\ C' \mid & \\
 & \quad \vdash\ [P]\ \texttt{WHILE}\ S\ \texttt{DO}\ C'\ [P \wedge \neg S]\} & (\text{While Rule \& } \vdash\ P \wedge S \Rightarrow\ E \geq 0) \\
\Rightarrow & \vdash\ [P]\ C\ [P \wedge \neg S] & \\
\Leftrightarrow & C \in [P,\ P \wedge \neg S] & (\text{Definition of } [P,\ P \wedge \neg S])
\end{array}$$

5.3 An example

The notation $[P_1,\ P_2,\ P_3,\ \cdots\ ,P_{n-1},\ P_n]$ will be used to abbreviate:

$$[P_1,\ P_2]\ ;\ [P_2,\ P_3]\ ;\ \cdots\ ;\ [P_{n-1},\ P_n]$$

The brackets around fully refined specifications of the form $\{C\}$ will be omitted – e.g. if $\mathcal{C}$ is a set of commands, then $R\ \texttt{:=}\ X\ ;\ \mathcal{C}$ abbreviates $\{R\ \texttt{:=}\ X\}\ ;\ \mathcal{C}$.

The familiar division program can be 'calculated' by the following refinement of the specification: $[Y > 0,\ X = R + (Y \times Q)\ \wedge\ R \leq Y]$

Let $\mathcal{I}$ stand for the invariant $X = R + (Y \times Q)$. In the refinement that follows, the comments in curley brackets after the symbol "$\supseteq$" indicate the refinement law used for the step.

$$
\begin{array}{l}
[Y > 0,\ \mathcal{I}\ \wedge\ R \leq Y] \\
\supseteq\ \text{(Sequencing)} \\
[Y > 0,\ R = X\ \wedge\ Y > 0,\ \mathcal{I}\ \wedge\ R \leq Y] \\
\supseteq\ \text{(Assignment)} \\
R \ \texttt{:=}\ X\ \texttt{;}\ [R = X\ \wedge\ Y > 0,\ \mathcal{I}\ \wedge\ R \leq Y] \\
\supseteq\ \text{(Sequencing)} \\
R \ \texttt{:=}\ X\ \texttt{;}\ [R = X\ \wedge\ Y > 0,\ R = X\ \wedge\ Y > 0\ \wedge\ Q = 0,\ \mathcal{I}\ \wedge\ R \leq Y] \\
\supseteq\ \text{(Assignment)} \\
R \ \texttt{:=}\ X\ \texttt{;}\ Q\ \texttt{:=}\ 0\ \texttt{;}\ [R = X\ \wedge\ Y > 0\ \wedge\ Q = 0,\ \mathcal{I}\ \wedge\ R \leq Y] \\
\supseteq\ \text{(Precondition Weakening)} \\
R \ \texttt{:=}\ X\ \texttt{;}\ Q\ \texttt{:=}\ 0\ \texttt{;}\ [\mathcal{I}\ \wedge\ Y > 0,\ \mathcal{I}\ \wedge\ R \leq Y] \\
\supseteq\ \text{(Postcondition Strengthening)} \\
R \ \texttt{:=}\ X\ \texttt{;}\ Q\ \texttt{:=}\ 0\ \texttt{;}\ [\mathcal{I}\ \wedge\ Y > 0,\ \mathcal{I}\ \wedge\ Y > 0\ \wedge\ \neg(Y \leq R)] \\
\supseteq\ \text{(While)} \\
R \ \texttt{:=}\ X\ \texttt{;}\ Q\ \texttt{:=}\ 0\ \texttt{;} \\
\texttt{WHILE}\ Y \leq R\ \texttt{DO}\ [\mathcal{I}\ \wedge\ Y > 0\ \wedge\ Y \leq R\ \wedge\ R = n, \\
\qquad\qquad\quad \mathcal{I}\ \wedge\ Y > 0\ \wedge\ R < n] \\
\supseteq\ \text{(Sequencing)} \\
R \ \texttt{:=}\ X\ \texttt{;}\ Q\ \texttt{:=}\ 0\ \texttt{;} \\
\texttt{WHILE}\ Y \leq R\ \texttt{DO}\ [\mathcal{I}\ \wedge\ Y > 0\ \wedge\ Y \leq R\ \wedge\ R = n, \\
\qquad\qquad\quad X = (R - Y) + (Y \times Q)\ \wedge\ Y > 0\ \wedge\ (R - Y) < n, \\
\qquad\qquad\quad \mathcal{I}\ \wedge\ Y > 0\ \wedge\ R < n] \\
\supseteq\ \text{(Derived Assignment)} \\
R \ \texttt{:=}\ X\ \texttt{;}\ Q\ \texttt{:=}\ 0\ \texttt{;} \\
\texttt{WHILE}\ Y \leq R\ \texttt{DO}\ [\mathcal{I}\ \wedge\ Y > 0\ \wedge\ Y \leq R\ \wedge\ R = n, \\
\qquad\qquad\quad X = (R - Y) + (Y \times Q)\ \wedge\ Y > 0\ \wedge\ (R - Y) < n] \\
\qquad\qquad\quad R\ \texttt{:=}\ R - Y \\
\supseteq\ \text{(Derived Assignment)} \\
R \ \texttt{:=}\ X\ \texttt{;}\ Q\ \texttt{:=}\ 0\ \texttt{;} \\
\texttt{WHILE}\ Y \leq R\ \texttt{DO}\ Q\ \texttt{:=}\ Q + 1\ \texttt{;}\ R\ \texttt{:=}\ R - Y
\end{array}
$$

5.4 General remarks

The 'Morgan style of refinement' illustrated here provides laws for systematically introducing structure with the aim of eventually getting rid of specification statements. This style has been accused of being "programming in the microscopic".

The 'Back style' is less rigidly top-down and provides a more flexible (but maybe also more chaotic) program development framework. It also emphasises and supports transformations that distribute control (e.g. going from sequential to parallel programs). General algebraic laws not specifically involving specification statements are used, for example:

$$C \quad = \quad \texttt{IF}\ S\ \texttt{THEN}\ C\ \texttt{ELSE}\ C$$

which can be used both to introduce and eliminate conditionals.

Both styles of refinement include large-scale transformations (data refinement and superposition) where a refinement step actually is a much larger change than a simple `IF` or `WHILE` introduction. However, this will not be covered here.

Chapter 6

Higher Order Logic

Higher order logic generalises first order logic by supporting λ-notation and allowing variables to range over functions and predicates. To preserve consistency it is typed. Various programming logics can be embedded in higher order logic.

Higher order predicate calculus (also called "higher order logic" and "simple type theory") uses the familiar notation of first order logic.

- "$P(x)$" means "x has property P",
- "$\neg t$" means "not t",
- "$t_1 \ \wedge \ t_2$" means "t_1 and t_2",
- "$t_1 \ \vee \ t_2$" means "t_1 or t_2",
- "$t_1 \Rightarrow t_2$" means "t_1 implies t_2",
- "$\forall x.\ t[x]$" means "for all x it is the case that $t[x]$",
- "$\exists x.\ t[x]$" means "for some x it is the case that $t[x]$",
- "$\exists! x.\ t[x]$" means "there is a unique x such that $t[x]$".

The difference is that in higher order logic, statements (or formulae) are regarded as boolean valued terms, i.e. terms whose value is one of the two truth values (or booleans) T or F. In the phrases just listed, t, t_1 and t_2 stand for arbitrary boolean terms, and $t[x]$ stands for some boolean term containing the variable x.

The structure of terms in higher order logic is more general that in the first order case. There are four kinds of terms; these will be explained in detail later, but here is a quick overview:

1. **Variables.** These are sequences of letters or digits beginning with a letter. For example: x, y, P.
2. **Constants.** Constants stand for fixed values. They will be distinguished from variables by being either mathematical characters or strings in $\mathsf{sans\ serif}$ font. Examples of constants are: T, F (the truth-values), 0, 1, 2, ... (numbers), $+$, $\times$, (arithmetical operators) $<$, $\leq$ (arithmetical predicates).

3. **Function applications.** These have the general form $t_1\ t_2$ where t_1 and t_2 are terms, an example is $P\ 0$. Brackets can be inserted around terms to increase readability or to enforce grouping, thus $P\ 0$ is equivalent to $P(0)$. Binary function constants can be infixed. Thus one can write $t_1 + t_2$ instead of $+\ t_1\ t_2$.

4. **Lambda-terms.** These denote functions and have the form $\lambda x.\ t$ (where x is a variable and t a term). For example, $\lambda n.\ n + 1$ denotes the successor function.

The various kinds of statements are just terms in higher order logic. For example, $\neg t$ is just the application of the constant $\neg$ to the term t and $t_1 \Rightarrow t_2$ is just the infixed application of the binary constant $\Rightarrow$ to argument terms t_1 and t_2.

The representation of quantifiers as terms is less straightforward, because of bound variables. In higher order logic, there is only a single variable binding mechanism: λ-abstraction. The quantifiers $\forall$ and $\exists$ are regarded as constants and the quantifications $\forall x.\ t$ and $\exists x.\ t$ are 'user friendly syntax' for the terms $\forall(\lambda x.\ t)$ and $\exists(\lambda x.\ t)$. Such constants are called *binders*.

Example: $\forall n.\ P(n) \Rightarrow P(n+1)$ is written instead of $\forall(\lambda n. \Rightarrow(P(n))(P(+\ n\ 1)))$
$\Box$

Higher order logic generalizes first order logic by allowing *higher order* variables — i.e. variables ranging over functions and predicates. For example, the induction axiom for natural numbers can be written as:

$$\forall P.\ P(0) \land (\forall n.\ P(n) \Rightarrow P(n+1)) \Rightarrow \forall n.\ P(n)$$

and the legitimacy of simple recursive definitions (the Peano-Lawvere Axiom [30]) can be expressed by:

$$\forall n_0.\ \forall f.\ \exists! s.\ (s(0) = n_0) \land (\forall n.\ s(n+1) = f(s(n)))$$

Sentences like these are not allowed in first order logic: in the first example above P ranges over predicates; in the second example f and s range over functions.

An example that will be central to the next chapter is the representation of partial correctness specifications in higher order logic. This is done by defining a predicate (i.e. a function whose result is a truth value) Spec by:

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

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:

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

The sequencing rule in Hoare logic 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, which will be fully explained in the next chapter, make essential use of higher order variables; they can't be expressed in first-order logic.

6.1 Terms

The four kinds of terms in the higher order logic are *variables*, *constants*, *applications* (of a function to an argument) and *abstractions* (also called λ-terms). These are described in detail below.

6.1.1 Variables and constants

Variables and constants stand for values. Variables will normally be strings of letters and digits starting with a letter. They will be written in italics. Constants can either be strings of letters and digits starting with a letter but they will be written in sans serif font. In addition, there are some special non-alphanumeric symbols for the constants representing logical operators; these include: the equals sign ($=$), the equivalence symbol ($\equiv$), the negation symbol ($\neg$), the conjunction symbol ($\wedge$), the disjunction symbol ($\vee$), the implication symbol ($\Rightarrow$), the universal quantifier ($\forall$), the existential quantifier ($\exists$), the unique existence quantifier ($\exists!$) and Hilbert's epsilon symbol (ε). Other allowed constant symbols are the pairing symbol (comma: ,), the numerals 0, 1, 2 etc., the arithmetic functions $+$, $-$, $\times$ and $/$, and the arithmetic relations $<$, $>$, $\leq$ and $\geq$.

6.1.2 Function applications

Terms of the form $t_1(t_2)$ are called *applications* or *combinations*. The subterm t_1 is called the *operator* (or *rator*) and the term t_2 is called the *operand* (or *rand* or *argument*). The result of such a function application can itself be a function and thus terms like $(t_1(t_2))(t_3)$ are allowed. Functions that take functions as arguments or return functions as results are called *higher order*.

To save writing brackets, function applications can be written as $f\ x$ instead of $f(x)$. More generally we adopt the usual 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.

6.1.3 Lambda-terms

The version of higher order logic presented here[1] provides *lambda-terms* (also called *λ-terms* or *abstractions*) for denoting functions. Such a term has the form $\lambda x.\ t$

[1] The version of higher order logic presented here is based on 'Simple Type Theory' which was invented by the logician Alonzo Church in 1940 [8].

(where t is a term) and denotes the function f defined by:

$$f(x) \;=\; t$$

For example, $\lambda n.\ \mathsf{cos}(\mathsf{sin}(n))$ denotes the function f such that:

$$f(n) \;=\; \mathsf{cos}(\mathsf{sin}(n))$$

thus: $f(1) = \mathsf{cos}(\mathsf{sin}(1))$, $f(2) = \mathsf{cos}(\mathsf{sin}(2))$ etc. The variable x and term t are called respectively the *bound variable* and *body* of the λ-expression $\lambda x.\ t$. An occurrence of the bound variable in the body is called a *bound* occurrence. If an occurrence is not bound it is called *free*.

6.2 Types

The increased expressive power gained by allowing higher order variables is dangerous. Consider the predicate P defined by:

$$\mathsf{P}\ x \;=\; \neg(x\ x)$$

from this definition it follows that:

$$\mathsf{P}\ \mathsf{P} \;=\; \neg(\mathsf{P}\ \mathsf{P})$$

which is a version of Russell's paradox. Russell invented a method for preventing such inconsistencies based on the use of *types* [18]. The formulation used here is a simplification of Russell's type system due to Church [8] with extensions developed by Milner [15]. It is very similar to the type system of the ML programming language.

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

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

these denote the sets of booleans, individuals, natural numbers and real numbers 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:

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

where *list* is a unary type operator and $\to$ and $\times$ are an infixed binary type operators. The type $\sigma\ \mathit{list}$ denotes the set of lists of values of type σ, the type $\sigma_1{\to}\sigma_2$ denotes the set of functions with domain denoted by σ_1 and range denoted by σ_2 and the type $\sigma_1{\times}\sigma_2$ denotes the Cartesian product type of pairs whose first component has type σ_1 and second component has type σ_2 (see Section 6.3.3). In general, compound types are expressions of the form:

$$(\sigma_1,\ \ldots\ ,\sigma_n)op$$

where *op* is a type operator and $\sigma_1, \ldots, \sigma_n$ are types. If the operator has only one argument then the brackets can be omitted (hence σ *list*); the types $\sigma_1 {\to} \sigma_2$ and $\sigma_1 {\times} \sigma_2$ are *ad hoc* abbreviations for $(\sigma_1, \sigma_2)\mathit{fun}$ and $(\sigma_1, \sigma_2)\mathit{prod}$, respectively. We will use lower case *slanted* identifiers 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:
   (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 $t{:}\sigma$ 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:

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

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 [33], 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, $\mathit{state} \times \mathit{state} {\to} \mathit{bool}$ means $(\mathit{state} \times \mathit{state}) {\to} \mathit{bool}$. 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 the "." after 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 $(\lambda b.\ b) = (\lambda x.\ \mathsf{T})$.

6.2.1 Type variables and polymorphism

Consider the function twice defined by:

$$\mathsf{twice} \;=\; \lambda f.\ \lambda x.\ f(f(x))$$

If f is a function then $\mathsf{twice}(f)$, the result of applying twice to f, is the function $\lambda x.\ f(f(x))$; twice is thus a function-returning function, i.e. it is higher order. For example, if sin is a trigonometric function with type $\mathit{real}{\rightarrow}\mathit{real}$, then $\mathsf{twice}(\mathsf{sin})$ is $\lambda x.\ \mathsf{sin}(\mathsf{sin}(x))$ which is the function taking the sin of the sin of its argument, a function of type $\mathit{real}{\rightarrow}\mathit{real}$, and if not is a boolean function with type $\mathit{bool}{\rightarrow}\mathit{bool}$, then $\mathsf{twice}(\mathsf{not})$ is $\lambda x.\ \mathsf{not}(\mathsf{not}(x))$ which is the function taking the double negation of its argument, a function of type $\mathit{bool}{\rightarrow}\mathit{bool}$. What then is the type of the function twice? Since $\mathsf{twice}(\mathsf{sin})$ has type $\mathit{real}{\rightarrow}\mathit{real}$ it would appear that twice has the type $(\mathit{real}{\rightarrow}\mathit{real}){\rightarrow}(\mathit{real}{\rightarrow}\mathit{real})$. However, since $\mathsf{twice}(\mathsf{not})$ has type $\mathit{bool}{\rightarrow}\mathit{bool}$ it would also appear that twice has the type $(\mathit{bool}{\rightarrow}\mathit{bool}){\rightarrow}(\mathit{bool}{\rightarrow}\mathit{bool})$. Thus twice would appear to have two different types. In Church's Simple Type Theory this would not be allowed and we would have to define two functions, $\mathsf{twice}_{(\mathit{real}\rightarrow\mathit{real})\rightarrow(\mathit{real}\rightarrow\mathit{real})}$ and $\mathsf{twice}_{(\mathit{bool}\rightarrow\mathit{bool})\rightarrow(\mathit{bool}\rightarrow\mathit{bool})}$ say. In the version of higher order used here, *type variables* are used to overcome this messiness; for example, if α is a type variable then twice can be given the type $(\alpha{\rightarrow}\alpha){\rightarrow}(\alpha{\rightarrow}\alpha)$ and then it behaves as though it has all instances of this that can be obtained by replacing α by a type. Types containing type variables are called *polymorphic*, ones not containing variables are *monomorphic*. We shall call a term polymorphic or monomorphic if its type is polymorphic or monomorphic respectively. We will use α, β, γ etc. for type variables. This use of type variables is an extension of Church's simple type theory. It is due to Robin Milner and was developed by him for a special purpose logic called PPLAMBDA[15].

An *instance* of a type σ is a type obtained by replacing zero or more type variables in σ by types. Here are some instances of $(\alpha{\rightarrow}\alpha){\rightarrow}(\alpha{\rightarrow}\alpha)$:

$$(\mathit{real}{\rightarrow}\mathit{real}){\rightarrow}(\mathit{real}{\rightarrow}\mathit{real})$$
$$(\mathit{bool}{\rightarrow}\mathit{bool}){\rightarrow}(\mathit{bool}{\rightarrow}\mathit{bool})$$
$$((\alpha{\rightarrow}\mathit{bool}){\rightarrow}(\alpha{\rightarrow}\mathit{bool})){\rightarrow}((\alpha{\rightarrow}\mathit{bool}){\rightarrow}(\alpha{\rightarrow}\mathit{bool}))$$

In these examples α has been replaced by real, bool and $\alpha{\rightarrow}\mathit{bool}$ respectively. The only instances of monomorphic types are themselves.

All constants are assumed to have a fixed type. If this type is polymorphic then for the purposes of type checking the constant behaves as though it is assigned every instance of the type. For example, if twice had type $(\alpha{\rightarrow}\alpha){\rightarrow}(\alpha{\rightarrow}\alpha)$, then the terms $\mathsf{twice}(\mathsf{sin})$ and $\mathsf{twice}(\mathsf{not})$ would be well-typed.

6.3 Special Syntactic Forms

Certain applications are conventionally written in special ways, for example:

- $+\ t_1\ t_2$ is written $t_1 + t_2$
- $,\ t_1\ t_2$ is written (t_1, t_2)
- $\forall(\lambda x.\ t)$ is written $\forall x.\ t$

Constants can have a special syntactic status to support such forms. For example, $+$ and , are examples of *infixes* and $\forall$ is an example of a *binder*. Some other *ad hoc* syntactic forms are also allowed, these are explained below.

6.3.1 Infixes

Constants with types of the form $\sigma_1{\rightarrow}\sigma_2{\rightarrow}\sigma_3$ can be *infixes*. If f is an infixed constant then applications are written as $t_1\ \mathsf{f}\ t_2$ rather than as $\mathsf{f}\ t_1\ t_2$. Standard examples of infixes are the arithmetic functions $+$, $\times$ etc. Whether a constant is an infix or not has no logical significance, it is merely syntactic.

Examples of infixes are the following constants:

$$\begin{array}{lll} \wedge & : \mathit{bool}{\rightarrow}\mathit{bool}{\rightarrow}\mathit{bool} & \text{(Conjunction - i.e. ``and'')} \\ \vee & : \mathit{bool}{\rightarrow}\mathit{bool}{\rightarrow}\mathit{bool} & \text{(Disjunction - i.e. ``or'')} \\ \Rightarrow & : \mathit{bool}{\rightarrow}\mathit{bool}{\rightarrow}\mathit{bool} & \text{(Implication - i.e. ``implies'')} \\ \equiv & : \mathit{bool}{\rightarrow}\mathit{bool}{\rightarrow}\mathit{bool} & \text{(Equivalence - i.e. ``if and only if'')} \end{array}$$

Equality is also an infixed constant; it is polymorphic:

$$= : \alpha{\rightarrow}\alpha{\rightarrow}\mathit{bool}$$

Equivalence ($\equiv$) is equality ($=$) restricted to booleans. The constants $\wedge$, $\vee$, $\Rightarrow$, $\equiv$ and $=$ are all infixes.

6.3.2 Binders

It is sometimes more readable to write $\mathsf{f}\ x.\ t$ instead of $\mathsf{f}(\lambda x.\ t)$. For example, the quantifiers $\forall$ and $\exists$ are polymorphic constants:

$$\begin{array}{l} \forall : (\alpha{\rightarrow}\mathit{bool}){\rightarrow}\mathit{bool} \\ \exists : (\alpha{\rightarrow}\mathit{bool}){\rightarrow}\mathit{bool} \end{array}$$

The idea is that if $P : \sigma{\rightarrow}\mathit{bool}$, then $\forall(P)$ is true if $P(x)$ is true for all x and $\exists(P)$ is true if $P(x)$ is true for some x. Instead of writing $\forall(\lambda x.\ t)$ and $\exists(\lambda x.\ t)$ it is nice to be able to use the more conventional forms $\forall x.\ t$ and $\exists x.\ t$.

Any constant f with a type of the form $(\sigma_1{\rightarrow}\sigma_2){\rightarrow}\sigma_3$ can be a *binder*, so that instead of writing:

$$\mathsf{f}(\lambda x_1.\ \mathsf{f}(\lambda x_2.\ \cdots\ \mathsf{f}(\lambda x_n.\ t)\ \cdots\ ))$$

one writes:

$$\mathsf{f}\ x_1\ \cdots\ x_n.\ t$$

As with infixes, the binder status of a constant is purely syntactic.

Recall the statement of mathematical induction:

$$\forall P.\ P(0) \land (\forall n.\ P(n) \Rightarrow P(n+1)) \Rightarrow \forall n.\ P(n)$$

This is a term of type *bool*. Without using infix and binder notation it would be much less readable, namely:

$$\forall(\lambda P.\ \Rightarrow(\land(P\ 0)(\forall(\lambda n.\ \Rightarrow(P\ n)(P(+\ n\ 1)))))(\forall(\lambda n.\ Pn)))$$

The quantifiers $\forall$ and $\exists$ need not be primitive; they can be defined in terms of more primitive notions. We will not, however, go into this here.

6.3.3 Pairs and tuples

A function of n arguments can be represented as a higher order function of 1 argument that returns a function of n-1 arguments. Thus $\lambda m.\ \lambda n.\ m^2 + n^2$ represents the 2 argument function that sums the squares of its arguments. Functions of this form are called *curried*. An alternative way of representing multiple argument functions is as single argument functions taking *tuples* as arguments. To handle tuples a binary type operator *prod* is used. If $t_1{:}\sigma_1$ and $t_2{:}\sigma_2$ then the term (t_1, t_2) has type $(\sigma_1, \sigma_2)prod$ and denotes the pair of values. The type $(\sigma_1, \sigma_2)prod$ can also be written as $\sigma_1{\times}\sigma_2$. Another representation of the sum-squares function would be as a constant, sumsq say, of type $(num{\times}num){\to}num$ defined by:

$$\mathsf{sumsq}(m, n)\ =\ m^2 + n^2$$

A term of the form (t_1, t_2) is equivalent to the term $,\ t_1\ t_2$ where "," is a polymorphic infixed constant of type $\alpha{\to}\beta{\to}(\alpha{\times}\beta)$. Instead of having tuples as primitive, they will be treated as iterated pairs. Thus the term:

$$(t_1, t_2,\ \ldots\ , t_{n-1}, t_n)$$

is an abbreviation for:

$$(t_1, (t_2,\ \ldots\ , (t_{n-1}, t_n)\ \ldots\ ))$$

i.e. "," associates to the right. To match this, the infixed type operator $\times$ also associates to the right so that if $t_1{:}\sigma_1$, …, $t_n{:}\sigma_n$ then:

$$(t_1,\ \ldots\ , t_n) : \sigma_1{\times}\ \cdots\ {\times}\sigma_n$$

The type operator *prod* can be defined in terms of *fun* and thus pairing need not be primitive. We shall not go into this here.

6.3.4 Lists

To represent lists, types of the form σ *list* are used, together with constants Nil and Cons of types α *list* and $\alpha{\to}(\alpha\ list){\to}(\alpha\ list)$ respectively. A term with type σ *list* denotes a list of values all of type σ. Nil is the empty list; $[\,]$ is an alternative form of Nil and $[t_1;\cdots;t_n]$ as an alternative form for $\mathsf{Cons}\ t_1(\mathsf{Cons}\ t_2\cdots(\mathsf{Cons}\ t_n\ \mathsf{Nil})\cdots))$.

The difference between lists and tuples is:

1. Different lists of a given type can contain different numbers of elements, but all tuples of a given type contain exactly the same numbers of elements.
2. The elements of a list must all have the same type but elements of tuples can have different types.

6.3.5 Conditionals

The constant Cond is defined so that $\mathsf{Cond}\ t\ t_1\ t_2$ means "if t then t_1 else t_2". The special syntax $(t \to t_1 \mid t_2)$ is provided for such terms. The original conditional notation due to McCarthy used "," instead of "|".

6.3.6 Hilbert's ε-operator

If $t[x]$ is a boolean term containing a free variable x of type σ, then the Hilbert-term $\varepsilon x.\ t[x]$ denotes some value of type σ, a say, such that $t[a]$ is true. For example, the term $\varepsilon n.\ n < 10$ denotes some unspecified number less than 10 and the term $\varepsilon n.\ (n^2 = 25) \wedge (n \geq 0)$ denotes 5.

If there is no a of type σ such that $t[a]$ is true then $\varepsilon x.\ t[x]$ denotes a fixed but unspecified value of type σ. For example, $\varepsilon n.\ \neg(n = n)$ denotes an unspecified number. The logical axiom

$$\vdash\ \forall P\ x.\ P\ x \Rightarrow P(\varepsilon\ P)$$

which is called `SELECT_AX`, defines the meaning of ε. This axiom is equivalent to:

$$\vdash\ \forall P.\ (\exists x.\ P\ x) \Rightarrow P(\varepsilon\ P)$$

It must be admitted that the ε-operator looks rather suspicious. For a thorough discussion of it see [26]. It is useful for naming things one knows to exist but have no name. For example, the Peano-Lawvere axiom asserts that given a number n_0 and a function $f{:}num{\to}num$, there exists a unique sequence s defined recursively by:

$$(s(0) = n_0)\ \wedge\ (\forall n.\ s(n+1) = f(s(n)))$$

Using the ε-operator we can define a function, Rec say, that returns s when given the pair (n_0, f) as an argument:

$$\mathsf{Rec}(n_0, f) \quad = \quad \varepsilon s.\ (s(0) = n_0) \wedge (\forall n.\ s(n+1) = f(s(n)))$$

$\mathsf{Rec}(n_0, f)$ denotes the unique sequence whose existence is asserted by the Peano-Lawvere Axiom. It follows from this axiom that:

$$(\mathsf{Rec}(n_0, f)0 \ = \ n_0) \ \wedge \ (\forall n.\ \mathsf{Rec}(n_0, f)(n+1) = f(\mathsf{Rec}(n_0, f)n))$$

Many things that are normally primitive can be defined using the ε-operator. For example, the conditional term $\mathsf{Cond}\ t\ t_1\ t_2$ (meaning "if t then t_1 else t_2") can be defined by:

$$\mathsf{Cond}\ t\ t_1\ t_2 \quad = \quad \varepsilon x.\ ((t = \mathsf{T}) \Rightarrow (x = t_1)) \ \wedge \ ((t = \mathsf{F}) \Rightarrow (x = t_2))$$

One can use the ε-operator to simulate λ-abstraction: if the variable f does not occur in the term t, then the function $\lambda x.\ t$ is equivalent to $\varepsilon f.\ \forall x.\ f(x) = t$ ("the function f such that $f(x) = t$ for all x"). This idea can be used to create functional abstractions that cannot be expressed with simple λ-terms. For example, the factorial function is denoted by:

$$\varepsilon f.\ \forall n.\ (f(0) = 1) \quad \wedge \quad (f(n+1) = (n+1) \times f(n))$$

Terms like this can be used to simulate the kind of pattern matching mechanisms found in programming languages like ML.

The inclusion of ε-terms in the logic 'builds in' the Axiom of Choice [18]. In Set Theory, the Axiom of Choice states that if $\mathcal{S}$ is a family of sets then there exists a function, Choose say, such that for each non-empty $X \in \mathcal{S}$ we have $\mathsf{Choose}(X) \in X$. As sets are not primitive in higher order logic, we must reformulate Choose to act on the characteristic functions of sets rather than sets themselves. The characteristic function of a set X is the function f_X with range $\{\mathsf{T}, \mathsf{F}\}$ defined by $f_X(x) = \mathsf{T}$ if and only if $x \in X$. If P is any function with range $\{\mathsf{T}, \mathsf{F}\}$, we call P *non-empty* if for some x it is the case that $P(x) = \mathsf{T}$ (so f_X is non-empty if and only if X is non-empty). The higher order logic version of the Axiom of Choice asserts that there exists a function, Select say, such that if P is a non-empty function with range $\{\mathsf{T}, \mathsf{F}\}$ then $P(\mathsf{Select}(P)) = \mathsf{T}$. Intuitively $\mathsf{Select}\ P$ is just $\mathsf{Choose}\{x \mid P\ x = \mathsf{T}\}$.

Hilbert's ε-operator is a binder that denotes Select. More precisely ε is a binder with type $(\alpha{\to}bool){\to}\alpha$ which is interpreted so that if P has type $\sigma{\to}bool$ then:

- $\varepsilon(P)$ denotes some fixed (but unknown) value x such that $P(x) = \mathsf{T}$ if such a value exists;

- if no such value exists (i.e. $P(x) = \mathsf{F}$ for all x) then $\varepsilon(P)$ denotes some unspecified value in the set denoted by σ.

Having ε-terms forces every type to be non-empty because the term $\varepsilon x{:}\sigma.\mathsf{T}$ always denotes a member of σ.

6.4 Definitions

Definitions are axioms of the form $\vdash\ \mathsf{c} = t$ where c is a new constant and t is a closed term[2] (i.e. a term without any free variables) that doesn't contain c. Such a definition just introduces the constant c as an abbreviation for the term t. The requirement that c may not occur in t prevents definitions from being recursive, this is to rule out inconsistent 'definitions' like $\vdash\ \mathsf{c} = \mathsf{c} + 1$. A function definition:

$$\vdash\ \mathsf{f}\ =\ \lambda x_1\ \cdots\ x_n.\ t$$

can be written as:

$$\vdash\ \mathsf{f}\,x_1\ \cdots\ x_n\ =\ t$$

For example, the definition of Seq given in Section 7.3 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:

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

Definitions have the property that adding a new definition to the set of existing ones cannot introduce any new inconsistencies. As was shown by Russell and Whitehead [18], with suitable definitions, all of classical mathematics can be constructed from logic together with the assumption that there are infinitely many individuals (the Axiom of Infinity). It is thus not necessary to postulate axioms other than definitions.

6.5 Peano's axioms

The natural numbers are defined by introducing a type *num* and constants $0 : \mathit{num}$ and $\mathsf{Suc} : \mathit{num}{\to}\mathit{num}$ and then postulating Peano's Axioms[3]. These axioms are:

1. There is a number 0.
2. There is a function Suc called the successor function such that if n is a number then $\mathsf{Suc}\ n$ is a number.
3. 0 is not the successor of any number.
4. If two numbers have the same successor then they are equal.
5. If a property holds of 0 and if whenever it holds of a number then it also holds of the successor of the number, then the property holds of all numbers. This postulate is called *Mathematical Induction*.

[2]It is also necessary to require that all type variables occurring in the types of subterms of t also occur in the type of c, but this technicality is glossed over here.

[3]In fact, the type *num* and constants 0 and Suc can be defined in higher order logic and Peano's axioms then proved as theorems (given the Axiom of Infinity). We will not go into this here; in particular, we will not describe how types are defined.

The first two postulates hold because $0{:}\mathit{num}$ and $\mathsf{Suc}{:}\mathit{num}{\rightarrow}\mathit{num}$. The following axiom formalizes the third postulate:

$$\vdash\ \forall m.\ \neg(\mathsf{Suc}\ m = 0)$$

The forth postulate is:

$$\vdash\ \forall m\ n.\ (\mathsf{Suc}\ m = \mathsf{Suc}\ n) \Rightarrow (m = n)$$

The fifth postulate, Mathematical Induction, is higher order:

$$\vdash\ \forall P{:}\mathit{num}{\rightarrow}\mathit{bool}.\ P\ 0\ \wedge\ (\forall m.\ P\ m \Rightarrow P(\mathsf{Suc}\ m)) \Rightarrow \forall m.\ P\ m$$

The numerals 1, 2, 3 etc. are defined by:

$$\begin{array}{ll} \vdash & 1\ =\ \mathsf{Suc}\ 0 \\ \vdash & 2\ =\ \mathsf{Suc}(\mathsf{Suc}\ 0) \\ \vdash & 3\ =\ \mathsf{Suc}(\mathsf{Suc}(\mathsf{Suc}\ 0)) \\ & \bullet \\ & \bullet \\ & \bullet \end{array}$$

Because Suc is one-to-one these denote an infinite set of distinct values of type num.

6.5.1 Primitive recursion

The usual theorems of arithmetic can be derived from Peano's postulates. The first step in doing this is to provide a mechanism for defining functions recursively. For example, the usual 'definition' of + is:

$$\begin{array}{ll} \vdash & 0 + m\ =\ m \\ \vdash & (\mathsf{Suc}\ m) + n\ =\ \mathsf{Suc}(m + n) \end{array}$$

Unfortunately this isn't a definition. In order to convert such recursion equations into definitions we need the Primitive Recursion Theorem:

$$\begin{array}{ll} \vdash & \forall x{:}\alpha.\ \forall f{:}\alpha{\rightarrow}\mathit{num}{\rightarrow}\alpha.\ \exists \mathit{fun}{:}\mathit{num}{\rightarrow}\alpha. \\ & \quad (\mathit{fun}\ 0 = x)\ \wedge \\ & \quad (\forall m.\ \mathit{fun}(\mathsf{Suc}\ m) = f(\mathit{fun}\ m)m) \end{array}$$

The proof of this theorem from Peano's postulates is quite complicated and is omitted. To show that the Primitive Recursion Theorem solves the problem of defining + one specializes it by taking x to be $\lambda n.\ n$ and f to be $\lambda f'\ x'.\ \lambda n.\ \mathsf{Suc}(f'\ n)$, this yields:

$$\begin{array}{ll} \vdash & \exists \mathit{fun}.\ (\mathit{fun}\ 0 = (\lambda n.\ n))\ \wedge \\ & \qquad (\forall m.\ \mathit{fun}(\mathsf{Suc}\ m) = (\lambda f'\ x'.\ \lambda n.\ \mathsf{Suc}(f'\ n))\ (\mathit{fun}\ m)\ m) \end{array}$$

which is equivalent to:

$$\begin{array}{l}\vdash\ \exists fun.\ (fun\ 0\ n = n) \wedge \\ \qquad\qquad (fun(\mathsf{Suc}\ m)n = \mathsf{Suc}(fun\ m\ n))\end{array}$$

Thus, if we define $+$ by:

$$\begin{array}{l}\vdash\ +\ =\ \varepsilon fun.\ \forall m\ n.\ (fun\ 0\ n = n) \wedge \\ \qquad\qquad\qquad\qquad (fun(\mathsf{Suc}\ m)n = \mathsf{Suc}(fun\ m\ n))\end{array}$$

then it follows from the axiom for the ε-operator that:

$$\begin{array}{l}\vdash\ 0 + n\ =\ n \\ \vdash\ (\mathsf{Suc}\ m) + n\ =\ \mathsf{Suc}(m + n)\end{array}$$

as desired.

The method just used to define $+$ generalizes to any primitive recursive definition. Such a definition has the form:

$$\begin{array}{l}fun\ 0\ x_1\ \cdots\ x_n\ =\ f_1\ x_1\ \cdots\ x_n \\ fun\ (\mathsf{Suc}\ m)\ x_1\ \cdots\ x_n\ =\ f_2\ (fun\ m\ x_1\ \cdots\ x_n)\ m\ x_1\ \cdots\ x_n\end{array}$$

where fun is the function being defined and f_1 and f_2 are given functions. To define a fun satisfying these equations we first define:

$$\begin{array}{l}\vdash\ \mathsf{Prim\_Rec}\ =\ \lambda x\ f.\ \varepsilon fun.\ (fun\ 0 = x) \wedge \\ \qquad\qquad\qquad\qquad\qquad (\forall m.\ fun(\mathsf{Suc}\ m) = f(fun\ m)m)\end{array}$$

It then follows by the axiom for the ε-operator and the Primitive Recursion Theorem that:

$$\begin{array}{l}\vdash\ \mathsf{Prim\_Rec}\ x\ f\ 0\ =\ x \\ \vdash\ \mathsf{Prim\_Rec}\ x\ f\ (\mathsf{Suc}\ m)\ =\ f\ (\mathsf{Prim\_Rec}\ x\ f\ m)\ m\end{array}$$

A function fun satisfying the primitive recursive equations above can thus be defined by:

$$\vdash\ fun\ =\ \mathsf{Prim\_Rec}\ f_1\ (\lambda f\ m\ x_1\ \cdots\ x_n.\ f_2\ (f\ x_1\ \cdots\ x_n)\ m\ x_1\ \cdots\ x_n)$$

An example of a primitive recursion in this form is the definition of $+$:

$$\vdash\ +\ =\ \mathsf{Prim\_Rec}\ (\lambda x_1.\ x_1)\ (\lambda f\ m\ x_1.\ \mathsf{Suc}(f\ x_1))$$

6.5.2 Arithmetic

Standard arithmetic functions and relations can easily be defined. These include the primitive recursive infixes $+$, $-$, $\times$ and $>$ which are defined:

$$\vdash\ (0 + n = n)\ \wedge\ ((\mathsf{Suc}\ m) + n = \mathsf{Suc}(m + n))$$

$$\vdash \ (0 - n = 0) \ \wedge \ ((\mathsf{Suc}\ m) - n = ((m < n) \rightarrow 0 \mid \mathsf{Suc}(m - n)))$$
$$\vdash \ (0 \times n = 0) \ \wedge \ ((\mathsf{Suc}\ m) \times n = (m \times n) + n)$$
$$\vdash \ (0 > n = \mathsf{F}) \ \wedge \ ((\mathsf{Suc}\ m) > n = (m = n) \ \vee \ (m > n))$$

The division function is an infix / defined by:

$$\vdash \ m/n \ = \ \varepsilon x.\ m = n \times x$$

This satisfies:

$$\exists x.\ m = n \times x \ \vdash \ m = n \times (m/n)$$

The arithmetic relation $>$ is defined by primitive recursion above[4]. The other relations are defined without recursion by:

$$\vdash \ m < n \ = \ (n > m)$$
$$\vdash \ m \leq n \ = \ (m < n) \ \vee \ (m = n)$$
$$\vdash \ m \geq n \ = \ (m > n) \ \vee \ (m = n)$$

The various laws of arithmetic can be deduced from these definitions, together with Peano's axioms and axioms and rules of inference of predicate calculus.

6.5.3 Lists

Values of type σ *list* are finite lists of values of type σ. The two standard list processing functions:

$\mathsf{Nil} : \alpha\ \mathit{list}$
$\mathsf{Cons} : \alpha{\rightarrow}(\alpha\ \mathit{list}){\rightarrow}(\alpha\ \mathit{list})$

satisfy the the following primitive recursion theorem for lists:

$$\begin{aligned}&\vdash \ \forall x{:}\beta.\ \forall f{:}\beta{\rightarrow}\alpha{\rightarrow}(\alpha\ \mathit{list}){\rightarrow}\beta.\ \exists! fun{:}(\alpha\ \mathit{list}){\rightarrow}\beta.\\ &\qquad (fun\ \mathsf{Nil} = x) \wedge\\ &\qquad (\forall h\ t.\ fun(\mathsf{Cons}\ h\ t) = f(fun\ t)h\ t)\end{aligned}$$

This theorem, which can be proved from suitable definitions of Cons, Nil and the type operator *list* (not given here), serves as the only 'axiom' for lists. It can be used to define the additional list processing functions:

$\mathsf{Hd} : (\alpha\ \mathit{list}){\rightarrow}\alpha$
$\mathsf{Tl} : (\alpha\ \mathit{list}){\rightarrow}(\alpha\ \mathit{list})$
$\mathsf{Null} : (\alpha\ \mathit{list}){\rightarrow}\mathit{bool}$

[4] In fact, in the formal development of numbers, $<$ is usually defined using a higher order trick before primitive recusion has been established.

The definitions of these functions (which are left as an exercise for the reader) ensure that they have the usual properties, namely:

$$\begin{array}{l}
\vdash \ \mathsf{Null}\ \mathsf{Nil} \\
\vdash \ \forall x\ l.\ \neg(\mathsf{Null}(\mathsf{Cons}\ x\ l)) \\
\vdash \ \forall x\ l.\ \mathsf{Hd}(\mathsf{Cons}\ x\ l) = x \\
\vdash \ \forall x\ l.\ \mathsf{Tl}(\mathsf{Cons}\ x\ l) = l \\
\vdash \ \forall l.\ \neg(\mathsf{Null}\ l) \Rightarrow \mathsf{Cons}(\mathsf{Hd}\ l)(\mathsf{Tl}\ l) = l
\end{array}$$

In addition we want lists to have the following property, which is analogous to induction for numbers:

$$\vdash \ \forall P.\ (P\ \mathsf{Nil})\ \wedge\ (\forall l.\ (P\ l) \Rightarrow \forall x.\ P(\mathsf{Cons}\ x\ l)) \Rightarrow \forall l.\ P\ l$$

This property follows directly from the uniqueness part of the primitive recursion theorem for lists given above.

The following alternative notation for lists is allowed: the empty list Nil can be written as $[\,]$ and a list of the form $\mathsf{Cons}\ t_1(\mathsf{Cons}\ t_2 \cdots (\mathsf{Cons}\ t_n\ \mathsf{Nil})\cdots)$ can be written as $[t_1; \cdots ; t_n]$.

6.6 Semantics

In this section we give a very informal sketch of the intended set-theoretic semantics of higher order logic.

The essential idea is that types denote sets and terms denote members of these sets. Only well-typed terms are considered meaningful. If term t has type σ then t should denote a member of the set denoted by σ.

The meaning of a type depends on the interpretation of the type variables (as sets) that it contains. A type σ containing type variables $\alpha_1, \ldots, \alpha_m$ denotes a function from m-tuples of sets to sets, such a function is not itself a set but is a class. For example, the type $\alpha{\to}\alpha$ denotes the 'class function' that maps a set X to the set of functions from X to X (i.e. $\alpha{\to}\alpha$ denotes $X \mapsto \{f \mid f : X{\to}X\}$).

Polymorphic constants are interpreted as functions of the interpretations of the type variables in their type. For example, the standard meaning of the constant $\mathsf{I}{:}\alpha{\to}\alpha$ is the function that maps a set X (the interpretation of α) to the identity function on X.

The meaning of a term depends on the interpretation of the constants, free variables and type variables in it. The interpretation of a term t with type variables $\alpha_1, \ldots, \alpha_m$ and free variables $x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n$ is a function from $m{+}n$-tuples of sets to sets. More specifically, it is a function from tuples $(X_1, \ldots, X_m, v_1, \ldots, v_n)$ where each X_i is a set and each v_i is a member of the interpretation of σ_i (where σ_i is interpreted with respect to the interpretation of $\alpha_1, \ldots, \alpha_m$ as $X_1, \ldots, X_m$). For example, the interpretation of $(\lambda x{:}\alpha.\ x)\ y$ with respect to the tuple (X, v) is

v, where X is the interpretation of α and $v \in X$ is the interpretation of y (i.e. the term $(\lambda x{:}\alpha.\ x)\ y$ denotes $(X, v) \mapsto v$).

Type variables are regarded as implicitly universally quantified at the outermost level. Thus a theorem $\vdash\ (\lambda x{:}\alpha.\ x)\ y = y$ asserts that with respect to every interpretation of α as a (non-empty) set X the interpretation of $\lambda x{:}\alpha.\ x$ is a function which when applied to the interpretation, v say, of y yields v.

Chapter 7

Deriving Floyd-Hoare Logic

It is shown how Floyd-Hoare logic can be derived in higher order logic. This involves (i) defining the meaning of commands with suitable semantic definitions, (ii) regarding correctness specifications as notations for certain statements in higher order logic, and (iii) showing that these statements obey the laws of Floyd-Hoare logic. A number of alternative program specification methods are briefly discussed.

7.1 Semantic embedding

Specialized languages and logics can often be represented in higher order logic by the method of semantic embedding. To illustrate this consider the propositional language:

$$\mathit{wff} \ ::= \ \mathsf{True} \ | \ \mathsf{N}\ \mathit{wff} \ | \ \mathsf{C}\ \mathit{wff}\ \mathit{wff} \ | \ \mathsf{D}\ \mathit{wff}\ \mathit{wff}$$

One approach to embedding this little language, called deep embedding, is to represent *wff*s *inside* the host logic (higher order logic in this example) by values of some type, *wff* say, and then define *in the host logic* a semantic function, $\mathcal{M}$ say, by recursion:

$$\begin{array}{ll}
\mathcal{M}(\mathsf{True}) & = T \\
\mathcal{M}(\mathsf{N}\ w) & = \neg\mathcal{M}(w) \\
\mathcal{M}(\mathsf{C}\ w_1\ w_2) & = \mathcal{M}(w_1) \wedge \mathcal{M}(w_2) \\
\mathcal{M}(\mathsf{D}\ w_1\ w_2) & = \mathcal{M}(w_1) \vee \mathcal{M}(w_2)
\end{array}$$

Here $\mathcal{M}$ is a constant of higher order logic of type $\mathit{wff}\to\mathit{bool}$.

Another approach, called shallow embedding, is to set up notational conventions for translating *wff*s into host logic terms. Suppose $[\![w]\!]$ is the translation of w into higher order logic. The operation $w \mapsto [\![w]\!]$ is not defined 'inside' the host logic, but corresponds to an informal set of 'parsing and pretty-printing' conventions. Let $\leadsto$ mean "is translated by notational conventions to", then with shallow embedding:

$$\begin{array}{lll}
[\![\mathsf{True}]\!] & \leadsto & \texttt{"}T\texttt{"} \\
[\![\mathsf{N}\ w]\!] & \leadsto & \texttt{"}\neg\texttt{"} \frown [\![w]\!] \\
[\![\mathsf{C}\ w_1\ w_2]\!] & \leadsto & [\![w_1]\!] \frown \texttt{"}\wedge\texttt{"} \frown [\![w_2]\!] \\
[\![\mathsf{D}\ w_1\ w_2]\!] & \leadsto & [\![w_1]\!] \frown \texttt{"}\vee\texttt{"} \frown [\![w_2]\!]
\end{array}$$

Deep and shallow embedding are really two ends of a spectrum. At intermediate points of this spectrum some aspects of the semantics would be formalized inside the host logic and others as informal notational conventions.

The advantage of deep embedding is that theorems *about* the embedded language can be proved. For example:

$$\forall w_1\ w_2 \in \mathit{wff}.\ \mathcal{M}(\mathsf{C}w_1w_2) = \mathcal{M}(\mathsf{NDN}w_1\mathsf{N}w_2)$$

It formalizes more of the embedding, but also requires a host logic expressive enough to accomodate this formalization.

With shallow embedding only theorems *in* the embedded language are provable. In the example above, quantification over *wff*s is not expressible. There is less in the logic, and hence the embedding is less demanding on it and so it is often easier to support complex notations.

In the rest of this chapter, shallow embeddings of Hoare specifications will be described, together with an outline of how the axioms and rules of Floyd-Hoare logic can be derived.

7.2 A simple imperative programming language

We will only consider a subset of our little programming language; in particular, `FOR`-commands and arrays are omitted. Instead of having separate one and two-armed conditionals, we will instead just have two-armed conditionals togther with a `SKIP`-command. The syntax of this subset is specified by the BNF given below. In this specification, the variable N ranges over the *numerals* 0, 1, 2 etc, the variable V ranges over *program variables*[1] X, Y, Z etc, the variables E, E_1, E_2 etc. range over *integer expressions*, the variables B, B_1, B_2 etc. range over *boolean expressions* and the variables C, C_1, C_2 etc. range over *commands*.

$$E \quad ::= \quad N \ \mid\ V \ \mid\ E_1 + E_2 \ \mid\ E_1 - E_2 \ \mid\ E_1 \times\ E_2 \ \mid\ \ldots$$

$$B \quad ::= \quad E_1 {=} E_2 \ \mid\ E_1 \leq E_2 \ \mid\ \ldots$$

$$\begin{array}{lll} C & ::= & \texttt{SKIP} \\ & \mid & V\ \texttt{:=}\ E \\ & \mid & C_1\ \texttt{;}\ C_2 \\ & \mid & \texttt{IF}\ B\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2 \\ & \mid & \texttt{WHILE}\ B\ \texttt{DO}\ C \end{array}$$

A formal semantics of $\{P\}\ C\ \{Q\}$ in higher order logic will be given later.

[1] 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 7.3.

7.2.1 Axioms and rules of Hoare logic

Here are the axioms and rules of Floyd-Hoare logic for the language used in this chapter. These are minor variants of the ones given earlier. We write $\vdash \{P\}\ C\ \{Q\}$ if $\{P\}\ C\ \{Q\}$ 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 $\vdash P$, where P is a formula of predicate logic, if P can be deduced from the laws of logic and arithmetic.

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

A1: the `SKIP`-axiom. For any formula P:

$$\vdash \{P\}\ \texttt{SKIP}\ \{P\}$$

A2: the assignment-axiom. For any formula P, program variable V and integer expression E:

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

where $P\texttt{[}E/V\texttt{]}$ denotes the result of substituting E for all free occurrences of V in 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 $\vdash P' \Rightarrow P$ is a theorem of pure logic and $\vdash \{P\}\ C\ \{Q\}$ is a theorem of Hoare logic, then $\vdash \{P'\}\ C\ \{Q\}$ can be deduced by R1.

R1: the rule of precondition strengthening. For any formulae P, P' and Q, and command C:

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

R2: the rule of postcondition weakening. For any formulae P, Q and Q', and command C:

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

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 P, Q and R, and commands C_1 and C_2:

$$\frac{\vdash\ \{P\}\ C_1\ \{Q\} \qquad \vdash\ \{Q\}\ C_2\ \{R\}}{\vdash\ \{P\}\ C_1\,;\ C_2\ \{R\}}$$

R4: the `IF`-rule. For any formulae P, Q and B, and commands C_1 and C_2:

$$\frac{\vdash\ \{P\ \wedge\ B\}\ C_1\ \{Q\} \qquad \vdash\ \{P\ \wedge\ \neg B\}\ C_2\ \{Q\}}{\vdash\ \{P\}\ \texttt{IF}\ B\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2\ \{Q\}}$$

Notice that in this rule (and also in R5 below) it is assumed that 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 P and B, and command C:

$$\frac{\vdash\ \{P\ \wedge\ B\}\ C\ \{P\}}{\vdash\ \{P\}\ \texttt{WHILE}\ B\ \texttt{DO}\ C\ \{P\ \wedge\ \neg B\}}$$

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

7.3 Semantics in logic

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

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

Since `WHILE`-commands need not terminate, the functions denoted by commands will be *partial*. For example, for any state s and command C

$$\mathsf{Meaning}(\texttt{WHILE}\ \mathsf{T}\ \texttt{DO}\ C)(s)$$

will be undefined. Since functions in conventional predicate calculus are total, 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 C denotes c then:

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

i.e.

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

If c_{WHILE} is the predicate denoted by `WHILE` T `DO` 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:

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

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:

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

If e, b and c are the denotations of E, B and 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 $[\![E]\!]$ and $[\![B]\!]$ for the logic terms representing the denotations of E and 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 $[\![E]\!]$ and $[\![B]\!]$ are *terms*, i.e. syntactic objects.

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:

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

The convention adopted here is that upper case variables are program variables and lower case variables are logical variables (as in the example just given). 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

$T[X_1, \ldots, X_n]$ is a term of higher order logic whose upper case free variables of type *num* are $X_1, \ldots, X_n$ then

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

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

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

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

7.3.1 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 C, a term $[\![C]\!]$ of type $state \times state{\to}bool$ is defined as follows:

1. $[\![\texttt{SKIP}]\!] \ = \ \mathsf{Skip}$

   where the constant Skip is defined by:

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

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

   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:

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

3. $[\![C_1 \texttt{;} \ C_2]\!] \ = \ \mathsf{Seq}([\![C_1]\!], [\![C_2]\!])$

   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. $[\![\texttt{IF}\ B\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2]\!] \ = \ \mathsf{If}([\![B]\!], [\![C_1]\!], [\![C_2]\!])$

   where the constant If is defined by:

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

5. $[\![\texttt{WHILE}\ B\ \texttt{DO}\ C]\!] \ = \ \mathsf{While}([\![B]\!], [\![C]\!])$

   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:

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

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

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

$\square$

It might appear that by representing the meaning of commands with relations, we can give a semantics to nondeterministic constructs. For example, if $C_1 \parallel C_2$ is the nondeterministic choice 'either do C_1 or do C_2', then one might think that a satisfactory semantics would be given by:

$$[\![C_1 \parallel C_2]\!] \ = \ \mathsf{Choose}([\![C_1]\!], [\![C_2]\!])$$

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 $\forall t.\ t \lor \mathsf{F} = t$, it follows that:

$$\texttt{SKIP} \parallel \mathsf{c}_{WHILE} \;=\; \texttt{SKIP}$$

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 [36]; the example above shows that the relational semantics used here does not do it. This problem will appear again in connection with Dijkstra's theory of weakest preconditions in Section 7.6.2.

7.3.2 Semantics of partial correctness specifications

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

$$\forall s_1\ s_2.\ [\![P]\!]\ s_1 \ \land\ [\![C]\!](s_1, s_2) \Rightarrow [\![Q]\!]\ s_2$$

To abbreviate this formula, define a constant Spec by:

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

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

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

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.

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

The semantics of this is thus represented by the term:

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

which abbreviates:

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

$\square$

7.4 Floyd-Hoare logic as higher order logic

Floyd-Hoare logic can be embedded in higher order logic simply by regarding the concrete syntax given in Section 7.2 as an *abbreviation* for the corresponding semantic formulae described in Section 7.3. For example:

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

can be interpreted 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.

If partial correctness specifications are interpreted this way then, as shown in the rest of this section, the axioms and rules of Hoare logic 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. $\vdash\ \forall p\ v\ e.\ \mathsf{Spec}((\lambda s.\ p(\mathsf{Bnd}(e, v, s))), \mathsf{Assign}(v, e), p)$

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 rather tedious (and are omitted). 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 7.4.7).

7.4.1 Derivation of the SKIP-axiom

To derive the `SKIP`-axiom it must be shown for arbitrary P that:

$$\vdash \{P\}\ \texttt{SKIP}\ \{P\}$$

which abbreviates:

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

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

7.4.2 Derivation of precondition strengthening

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

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

Expanding abbreviations converts this to:

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

Specializing H3 yields:

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

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

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

Since X_1 , ... , 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:

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

Generalizing on the free variable s yields:

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

which is equivalent (by β-reduction) to

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

i.e.

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

7.4.3 Derivation of postcondition weakening

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

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

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

7.4.4 Derivation of the sequencing rule

To derive the sequencing rule, it must be shown that for arbitrary P, C_1, R, C_2 and and Q that:

$$\frac{\vdash\ \{P\}\ C_1\ \{Q\} \qquad \vdash\ \{Q\}\ C_2\ \{R\}}{\vdash\ \{P\}\ C_1\,;\ C_2\ \{R\}}$$

Expanding the abbreviations yields:

$$\frac{\vdash\ \mathsf{Spec}([\![P]\!],[\![C_1]\!],[\![Q]\!]) \qquad \vdash\ \mathsf{Spec}([\![Q]\!],[\![C_2]\!],[\![R]\!])}{\vdash\ \mathsf{Spec}([\![P]\!],\mathsf{Seq}([\![C_1]\!],[\![C_2]\!]),[\![R]\!])}$$

The validity of this rule follows directly from H5.

7.4.5 Derivation of the `IF`-rule

To derive the `IF`-rule, it must be shown that for arbitrary P, B, C_1, C_2 and and Q that:

$$\frac{\vdash\ \{P\ \wedge\ B\}\ C_1\ \{Q\} \qquad \vdash\ \{P\ \wedge\ \neg B\}\ C_2\ \{Q\}}{\vdash\ \{P\}\ \texttt{IF}\ B\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2\ \{Q\}}$$

Expanding abbreviations yields:

$$\frac{\vdash\ \mathsf{Spec}([\![P\ \wedge\ B]\!],[\![C_1]\!],[\![Q]\!]) \qquad \vdash\ \mathsf{Spec}([\![P\ \wedge\ \neg B]\!],[\![C_2]\!],[\![Q]\!])}{\vdash\ \mathsf{Spec}([\![P]\!],\mathsf{If}([\![B]\!],[\![C_1]\!],[\![C_2]\!]),[\![Q]\!])}$$

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

7.4.6 Derivation of the `WHILE`-rule

To derive the `WHILE`-rule, it must be shown that for arbitrary P, B and C that:

$$\frac{\vdash\ \{P\ \wedge\ B\}\ C\ \{P\}}{\vdash\ \{P\}\ \texttt{WHILE}\ B\ \texttt{DO}\ C\ \{P\ \wedge\ \neg B\}}$$

Expanding abbreviations yields:

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

This follows from H7.

7.4.7 Derivation of the assignment axiom

To derive the assignment axiom, it must be shown that for arbitrary P, E and V:

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

This abbreviates:

$$\vdash \ \mathsf{Spec}([\![P\,\texttt{[}E/V\texttt{]}]\!], \mathsf{Assign}(\text{`}V\text{`}, [\![E]\!]), [\![P]\!])$$

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 $[\![P]\!]$, 'V' and $[\![E]\!]$ yields:

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

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

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

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

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

Performing the substitution in the left hand side yields:

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

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

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

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 'V' $\neq X_i$):

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

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

results in:

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

which is true since:

$$[\![E[V, X_1, \ldots, X_n]]\!]s \;=\; E[s`V`, s`X_1`, \ldots, s`X_n`]$$

Although this derivation might appear tricky at first sight, it is straightforward and easily mechanized.

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. $P\,\texttt{[}E/V\texttt{]}$ 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 [6].

7.5 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 $\mathsf{Halts}([\![P]\!], [\![C]\!])$, 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:

$$\vdash\ \mathsf{Halts}([\![\mathsf{T}]\!],\ \mathsf{Choose}([\![\texttt{WHILE T DO SKIP}]\!],\ [\![\texttt{SKIP}]\!]))$$

($\|$ and Choose are described in Section 7.3). The meaning of $\mathsf{Halts}([\![P]\!], [\![C]\!])$ is 'some computation of C starting from a state satisfying P terminates' this is quite different from 'every computation of C starting from a state satisfying P terminates'. The latter stronger kind of termination requires a more complex kind of semantics for its formalization (e.g. one using powerdomains [36]). We will not pursue this here and assume that commands are as defined at the beginning of Section 7.2 (i.e. no parallel composition $\|$ in the language). In this language, termination is adequately formalized by Halts, so 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 $[P]\ C\ [Q]$ is represented by $\mathsf{Total\_Spec}([\![P]\!], [\![C]\!], [\![Q]\!])$.

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

T6 shows that if x is a *variant*, i.e. a variable whose value decreases each time 'around the loop' (n is assumed to be of natural number type, i.e. $n \geq 0$), then the `WHILE`-command halts.

7.5.1 Derived rules for total correctness

Using T1 – T6 above and H1 – H7 of Section 7.4, it is straightforward to apply the methods described in Section 7.4 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.

$$\vdash\ [P]\ \texttt{SKIP}\ [P]$$

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

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

$$\frac{\vdash\ [P]\ C_1\ [Q] \qquad \vdash\ [Q]\ C_2\ [R]}{\vdash\ [P]\ C_1\,;\ C_2\ [R]}$$

$$\frac{\vdash\ [P\ \wedge\ B]\ C_1\ [Q] \qquad \vdash\ [P\ \wedge\ \neg B]\ C_2\ [Q]}{\vdash\ [P]\ \texttt{IF}\ B\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2\ [Q]}$$

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.

$$\frac{\vdash\ [P\ \wedge\ B\ \wedge\ (\texttt{N} = n)]\ C\ [P\ \wedge\ (\texttt{N} < n)]}{\vdash\ [P]\ \texttt{WHILE}\ B\ \texttt{DO}\ C\ [P\ \wedge\ \neg B]}$$

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

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

The converse to this is only valid if C contains no `WHILE`-commands. We thus have the derived rule:

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

if C contains no `WHILE`-commands.

7.6 Other programming logic constructs

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

(i) VDM-style specifications.

(ii) Weakest preconditions.

(iii) Dynamic logic.

7.6.1 VDM-style specifications

The Vienna Development Method (VDM) [24]) 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:

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

could be written in a VDM-style as:

$$\{\mathsf{T}\}\ R\texttt{:=}\, X\,\texttt{;}\ X\texttt{:=}\, Y\,\texttt{;}\ Y\texttt{:=}\, R\ \{Y = \overleftharpoon{X}\ \wedge\ X = \overleftharpoon{Y}\}$$

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

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

can be thought of as an abbreviation for

$$\begin{array}{l}\{P[X_1,\ldots,X_n]\ \wedge\ X_1=\overleftarrow{X_1}\ \wedge\ \ldots\ \wedge\ X_n=\overleftarrow{X_n}\}\\ \ C\\ \{Q[X_1,\ldots,X_n,\overleftarrow{X_1},\ldots,\overleftarrow{X_n}]\}\end{array}$$

where $\overleftarrow{X_1}$, ... , $\overleftarrow{X_n}$ are distinct logical variables not occurring in C.

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 [24]. 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 postcondition as a binary relation on the initial and final states. This can be formalized by regarding

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

as an abbreviation for

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

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:

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

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 7.3:

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

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 [24]:

$$\frac{\vdash\ \{P\ \wedge\ B\}\ C\ \{P\ \wedge\ R\}}{\vdash\ \{P\}\ \mathtt{WHILE}\ B\ \mathtt{DO}\ C\ \{P\ \wedge\ \neg B\ \wedge\ R^*\}}$$

Where R^* is the reflexive closure of R defined by

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

and R^n is definable in higher order logic by the following primitive recursion:

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

Aczel pointed out that his version of the `WHILE`-rule can be converted into a rule of total correctness simply by requiring R to be transitive and well-founded:

$$\frac{\vdash\ [P\ \wedge\ B]\ C\ [P\ \wedge\ R] \qquad \vdash\ \mathsf{Transitive}\ R \qquad \vdash\ \mathsf{Well\_Founded}\ R}{\vdash\ [P]\ \mathtt{WHILE}\ B\ \mathtt{DO}\ C\ [P\ \wedge\ \neg B\ \wedge\ R^*]}$$

where:

$$\mathsf{Transitive}\ r \quad = \quad \forall s_1\ s_2\ s_3.\ r(s_1,s_2)\ \wedge\ r(s_2,s_3) \Rightarrow r(s_1,s_3)$$

$$\mathsf{Well\_Founded}\ r \quad = \quad \neg\exists f : num{\rightarrow}state.\ \forall n.\ r(f(n), f(n+1))$$

Whilst Transitive can be defined in first order logic, Well_Founded cannot: it needs the higher order quantification $\exists f$.

7.6.2 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[2].

In his book [10], 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 C is a command and Q a predicate, then:

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

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 $state{\rightarrow}bool$), 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:

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

The weakest predicate satisfying a condition can be given a general definition using Hilbert's ε-operator.

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

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 seem 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.

[2]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 7.3.1).

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

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

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

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

$$\vdash \ \mathsf{Total\_Spec}(p,c,q) \quad = \quad \forall s.\ p\ s \Rightarrow \mathsf{wp}(c,q)\ s$$

7.6.3 Strongest postconditions

Strongest postconditions are dual to weakest preconditions and go 'forwards' starting from a precondition whereas weakest preconditions go 'backwards' starting from a postcondition. The idea is that if C is a command and P a predicate, then:

- $\mathsf{sp}(C,P) \ = $ '*The strongest predicate Q such that $\{P\}\ C\ \{Q\}$*'

this is a partial correctness notion (we won't consider the total correctness version here). The strongest predicate satisfying a condition can be given a general definition using Hilbert's ε-operator.

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

The relationship between Hoare's notation and strongest postconditions is given by:

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

and one can prove that:

$$\vdash \ \mathsf{sp}(c,p) \quad = \quad \lambda s'.\ \exists s.\ p\ s \wedge c(s,s')$$

7.6.4 Verification using weakest preconditions and strongest postconditions

The weakest precondition and strongest postcondition can be 'calculated' using syntactic rules (in fact, it was these rules that came first, and only later was the semantic viewpoint developed). The rules are:

$$\begin{array}{ll} \mathsf{sp}(\mathtt{SKIP},P) & = P \\ \mathsf{wlp}(\mathtt{SKIP},Q) & = Q \end{array}$$

$$\begin{array}{ll} \mathsf{sp}((V := E),P) & = \exists v.\ (V{=}E[v/V]) \wedge P[v/V] \\ \mathsf{wlp}((V := E),Q) & = Q[E/V] \end{array}$$

$$\begin{array}{ll} \mathsf{sp}((S_1\,;S_2),P) & = \mathsf{sp}(S_2,(\mathsf{sp}(S_1,P))) \\ \mathsf{wlp}((S_1\,;S_2),Q) & = \mathsf{wlp}(S_1,(\mathsf{wlp}(S_2,Q))) \end{array}$$

$$\begin{array}{ll} \mathsf{sp}((\mathtt{IF}\,B\,\mathtt{THEN}\,S_1\,\mathtt{ELSE}\,S_2),P) & = (\mathsf{sp}(S_1,(P \wedge B)))\ \vee\ (\mathsf{sp}(S_2,(P \wedge \neg B))) \\ \mathsf{wlp}((\mathtt{IF}\,B\,\mathtt{THEN}\,S_1\,\mathtt{ELSE}\,S_2),Q) & = ((\mathsf{wlp}(S_1,Q)) \wedge B)\ \vee\ ((\mathsf{wlp}(S_2,Q)) \wedge \neg B) \end{array}$$

$$\begin{array}{ll} \mathsf{sp}((\mathtt{WHILE}\,B\,\mathtt{DO}\,S), P) & = (\mathsf{sp}((\mathtt{WHILE}\,B\,\mathtt{DO}\,S), (\mathsf{sp}(S, (P \wedge B))))) \ \vee \ (P \wedge \neg B) \\ \mathsf{wlp}((\mathtt{WHILE}\,B\,\mathtt{DO}\,S), Q) & = (\mathsf{wlp}(S, (\mathsf{wlp}((\mathtt{WHILE}\,B\,\mathtt{DO}\,S), Q))) \wedge B) \ \vee \ (Q \wedge \neg B) \end{array}$$

these rules operate directly on the syntax of commands and formulas, but using semantic embedding they are easily converted to equations in higher order logic. For example, weakest preconditions for straight line code are given by:

$$\vdash \ \mathsf{wlp}([\![\mathtt{SKIP}]\!],\ q) = q$$

$$\vdash \ \mathsf{wlp}([\![V\ \mathtt{:=}\ E]\!],\ q) = \lambda s.\ q(\mathsf{Bnd}\ ([\![E]\!]s)\ `V`\ s)$$

$$\vdash \ \mathsf{wlp}([\![C_1\ \mathtt{;}\ C_2]\!],\ q) = \mathsf{wlp}([\![C_1]\!], \mathsf{wlp}([\![C_2]\!], q))$$

$$\vdash \ \mathsf{wlp}([\![\mathtt{IF}\ B\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2]\!],\ q) = \lambda s.\ ([\![B]\!]s \rightarrow \mathsf{wlp}([\![C_1]\!], q)s \mid \mathsf{wlp}([\![C_2]\!], q)s)$$

Exercise 54
Formulate the corresponding semantic equations for the strongest postconditions of straight line code. $\Box$

For straight line code one can prove $\{P\}\ C\ \{Q\}$ by calculating $\mathsf{wlp}(C, Q)$ and then proving the formula $P \Rightarrow \mathsf{wlp}(C, Q)$. The method of verification conditions described earlier is closely related to this. The weakest precondition of `WHILE`-loops cannot be calculated as a finite formula in general, but an approach using invariants can be formulated. It is possible to use the calculation of wlp (or wp for total correctness) as the basis for a verification method that only requires `WHILE` and `FOR` loops to be annotated with invariants – annotations in the other positions we described are not needed.

Calculating strongest postconditions goes 'forwards' starting from the precondition. It is related to symbolic execution and is the basis for a number of automatic methods for checking correctness properties. The use of strongest postcondition calculation for full proof of correctness is not common (though not unknown) and may be an interesting research topic to pursue as a step towards combining automatic property checking with deductive proof of correctness.

7.6.5 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 [37, 14]. In dynamic logic, states of computation are thought of as *possible worlds*, and if a command 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 $\mathtt{[}C\mathtt{]}$ and $\mathtt{<}C\mathtt{>}$ for each command C. These can be defined on the relation c denotated by C as follows:

$$\begin{array}{lcl} \mathtt{[}c\mathtt{]}\,q & = & \lambda s.\ \forall s'.\ c(s,s') \Rightarrow q\ s' \\ \mathtt{<}c\mathtt{>}q & = & \neg(\mathtt{[}c\mathtt{]}(\neg q)) \end{array}$$

where $\neg$ is negation lifted to predicates (see preceding section). Noe that $\mathtt{[}c\mathtt{]}\,q$ is sematically the same as $\mathsf{wlp}(c,q)$.

A typical theorem of dynamic logic is:

$$\vdash\ \forall C\ q.\ \models \mathtt{<}[\![C]\!]\mathtt{>}q \Rrightarrow \mathtt{[}[\![C]\!]\mathtt{]}\,q$$

This is a version of the modal logic principle that says that if the accessibility relation is functional then $\Diamond q \Rightarrow \Box q$ [14].

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

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

$$\vdash\ (\models \mathtt{<}[\![C]\!]\mathtt{>}q)\ =\ \mathsf{Total\_Spec}(\mathsf{T}, [\![C]\!], q)$$

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

$$\vdash\ \mathsf{Total\_Spec}(p, [\![C]\!], q)\ =\ (\models p \Rrightarrow \mathtt{<}[\![C]\!]\mathtt{>}q)$$

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:

$$\vdash\ \mathsf{wlp}([\![C]\!], q)\ =\ \mathtt{[}[\![C]\!]\mathtt{]}\,q$$

$$\vdash\ \mathsf{wp}([\![C]\!], q)\ =\ \mathtt{<}[\![C]\!]\mathtt{>}q$$

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

Bibliography

[1] Alagić, S. and Arbib, M.A., *The Design of Well-structured and Correct Programs*, Springer-Verlag, 1978.

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

[3] Back, R.J.R, *On correct refinement of programs* in *Journal of Computer and Systems Sciences*, Vol. 23, No. 1, pp 49-68, August 1981.

[4] Backhouse, R.C., *Program Construction and Verification*, Prentice Hall, 1986.

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

[6] 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.

[7] Chang, C. and Lee, R.C., *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, 1973.

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

[9] 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.

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

[11] 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.

[12] Genesereth, M.R and Nilsson, N.J., *Logical Foundations of Artificial Intelligence*, Morgan Kaufman Publishers, Los Altos, 1987.

[13] Nagel, E. and Newman, J.R., *Gödel's Proof*, Routledge & Kegan Paul, London, 1959.

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

[15] M. Gordon, R. Milner and C. Wadsworth. *Edinburgh LCF: A mechanised logic of computation*. Lecture Notes in Computer Science No. 78, Springer-Verlag, 1979.

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

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

[18] W. Hatcher. *The Logical Foundations of Mathematics*. Pergamon Press, 1982.

[19] Hehner, E.C.R., *The Logic of Programming*, Prentice Hall, 1984.

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

[21] Hoare, C.A.R., 'A Note on the FOR Statement', BIT, **12**, pp. 334-341, 1972.

[22] Hoare, C.A.R., 'Programs are predicates', in Hoare, C.A.R. and Shepherdson, J.C. (eds), *Mathematical Logic and Programming Languages*, Prentice Hall, 1985.

[23] Jones, C.B., *Systematic Software Development Using VDM*, Prentice Hall, 1986.

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

[25] Joyce, E., 'Software bugs: a matter of life and liability', *Datamation*, **33**, No. 10, May 15, 1987.

[26] A. Leisenring. *Mathematical Logic and Hilbert's ε-Symbol*. Macdonald & Co. Ltd. London, 1969.

[27] Ligler, G.T., 'A mathematical approach to language design', in *Proceedings of the Second ACM Symposium on Principles of Programming Languages*, pp. 41-53, 1985.

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

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

[30] S. MacLane and G Birkhoff. *Algebra*. The Macmillan Company, 1967.

[31] Manna, Z., *Mathematical Theory of Computation*, McGraw-Hill, 1974.

[32] Manna, Z. and Waldinger, R., *The Logical Basis for Computer Programming*, Addison-Wesley, 1985.

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

[34] Morgan, C.C., *Programming from Specifications*, Prentice-Hall, 1990.

[35] Morris, J.M. *A Theoretical Basis for Stepwise Refinement and the Programming Calculus*, in *Science of Computer Programming*, vol. 9, pp 287–306, 1989.

[36] 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.

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

[38] Reynolds, J.C., *The Craft of Programming*, Prentice Hall, London, 1981.

[39] Schoenfield, J.R., *Mathematical Logic*, Addison-Wesley, Reading, Mass., 1967.