Background reading on *Hoare Logic*

Mike Gordon

Learning Guide for the CST Part II course. This document aims to provide background reading to support the lectures – think of it as a free downloadable textbook. Chapters 1–5 introduce classical ideas of specification and proof of programs due to Floyd and Hoare.[1] Although much of the material is old – see the dates on some of the cited references – it is still a foundation for current research. Chapter 6 is a very brief introduction to program refinement; this provides rules to 'calculate' an implementation from a Hoare-style specification. Chapter 7 is an introduction to the ideas of separation logic, an extension of Hoare logic for specifying and verifying programs that manipulate pointers. Separation logic builds on early ideas of Burstall, but its modern form is due to O'Hearn and Reynolds.

Note that there may be topics presented in the lectures that are not covered in this document and there may be material in this document that is not related to the topics covered in the lectures. For example, the topics of program refinement and separation logic may only be described very superficially, if at all. **The examination questions will be based on the material presented in the lectures.**

The Part II course *Hoare Logic* has evolved from an earlier Part II course, whose web page can be found on my home page (`www.cl.cam.ac.uk/~mjcg`). Some exam questions from that course might be good exercises (but note that some are based on material not covered in this course). A separate document containing exercises for the current course is available from the web page.

Warning. The material here consists of reorganized extracts from lecture notes for past courses, together with new material. There is a fair chance that notational inconsistencies, omissions and errors are present. If you discover such defects please send details to `Mike.Gordon@cl.cam.ac.uk`.

Acknowledgements. Thanks to Martin Vechev and John Wickerson for finding many errors (some serious) in a previous draft of these notes and also for suggestions for improving the text.

MJCG February 2, 2016

[1] Hoare Logic is sometimes called Floyd-Hoare Logic, due to the important contributions of Floyd to the underlying ideas.

Contents

Chapter 1

Program Specification

A simple programming language containing assignments, conditionals, blocks and `WHILE`*-loops is introduced. This is then used to illustrate Hoare's notation for specifying the partial correctness of programs. Hoare's notation uses formal logic notation to express conditions on the values of program variables. This notation is described informally and illustrated with examples.*

1.1 Introduction

In order to prove the correctness of a program mathematically 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*, which, until Chapter 7, we take to be the values of program variables. 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 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 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' [14]).

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 $\texttt{X} + \texttt{1}$, $\sqrt{2}$ etc. which denote values (usually numbers).

3. The symbols S, S_1, … , S_n stand for arbitrary statements. These are conditions like $\texttt{X} < \texttt{Y}$, $\texttt{X}^2 = \texttt{1}$ 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. All variables are assumed to have global scope.

Example: `X:=X+1`

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

1.2.2 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`.

1.2.3 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`.

1.2.4 `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.

1.2.5 Summary of syntax

The syntax of our little language can be summarised with the following specification in BNF notation[1]

```
<command>
  ::= <variable>:=<term>
   |  <command>; ... ;<command>
   |  IF <statement> THEN <command> ELSE  <command>
   |  WHILE <statement> DO <command>
```

Note that:

- *Variables*, *terms* and *statements* are as described in Section 1.5.
- The BNF syntax is ambiguous: for example, it does not specify whether `IF` S_1 `THEN` C_1 `ELSE` C_2`;` C_3 means `(IF` S_1 `THEN` C_1 `ELSE` C_2`);` C_3 or means `IF` S_1 `THEN` C_1 `ELSE (`C_2`;` C_3`)`. We will clarify, whenever necessary, using brackets.

1.2.6 Historical note

The old Part II course *Specification and Verification I* was based on a language similar to the one described above, but with additional features: blocks (with local variables), `FOR`-commands and arrays. Blocks and `FOR`-commands don't add fundamentally new ideas so they will not be covered; arrays are better handled using separation logic (see Section 7). In the old course I used `BEGIN` and `END` to group commands, whereas here I just use parentheses. Thus previously I would have written `BEGIN` C_1`;`C_2 `END` instead of `(`C_1`;`C_2`)`. I mention this as it is may help in reusing old examination questions as exercises for this course.

1.3 Hoare's notation

In a seminal paper [13] C.A.R. Hoare introduced the notation[2] $\{P\}\ C\ \{Q\}$, which is sometimes called a *Hoare triple*, for specifying what a program does. In such a Hoare triple:

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

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

- 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 true.

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* C 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 two conditions apply:

(i) If C is executed in a state satisfying P, then 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, the unsolved Collatz conjecture is related to whether the program below terminates for all values of `X` (see the exercise below):

```
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, though since the `ELSE`-arm of the conditional here is only taken if `X` is even, no rounding is actually needed.)

The famous mathematician Paul Erdös said about the Collatz conjecture: “Mathematics is not yet ready for such problems.” He offered $500 for its solution.[3]

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

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

This says that if the execution of `R:=X; X:=Y; Y:=R` 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 *logical*, *auxiliary* or *ghost* variables.

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

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

[3] `http://en.wikipedia.org/wiki/Collatz_conjecture`

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

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).

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.

The following are examples of *atomic statements*.

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

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

Statements are built out of *terms* like:

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

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

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

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

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

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

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

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

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

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:

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

Hoare logic

The idea of formal proof is discussed. Hoare logic (also called 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 × 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 `1`.
5.	$(\mathtt{X}+\mathtt{1})\times\mathtt{X}$	$= \mathtt{X}\times\mathtt{X}+\mathtt{1}\times\mathtt{X}$	Right distributive law of × 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 `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 Hoare logic provides. The formulation of the deductive system is due to Hoare [13], but some of the underlying ideas originated with Floyd [9].

A proof in 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.	$\sqrt{-1 \times -1}$	$= \sqrt{-1 \times -1}$	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. 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.

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. There was a legal action resulting from the death of several people due to radiation overdoses by a cancer treatment machine that had a software bug [15]. Formal proof of correctness provides a way of establishing the absence of bugs when exhaustive testing is impossible (as it almost always is).

The Hoare deductive system for reasoning about programs will be explained and illustrated. The mathematical analysis of the soundness and completeness of the system is discussed in Section 4.

2.1 Axioms and rules of 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 Hoare logic) then we write $\vdash S$ to mean that S has a proof. The statements that have proofs are called *theorems*. As discussed earlier, in these notes only the axioms and rules of inference for 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 are assumed known (e.g. from the *Logic and Proof* course).

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

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

This says the *conclusion* $\vdash S$ may be deduced from the $\vdash S_1, \ldots, \vdash S_n$, which are the *hypotheses* of the rule. The hypotheses can either all be theorems of Hoare logic (as in the sequencing rule below), or a mixture of theorems of 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 formalise 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`[`E/V`]` to mean the result of replacing all occurrences of V in P by E. Read P`[`E/V`]` as 'P with E for V'.

For example,

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

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

$$V\mathtt{[}E/V\mathtt{]} \;=\; E$$

which is analogous to the cancellation property of fractions

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

The Hoare assignment axiom

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

Where V is any variable, E is any expression, P is any statement and the notation $P\mathtt{[}E/V\mathtt{]}$ 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. $\vdash\; \{\mathtt{Y} = 2\}\; \mathtt{X} := 2\; \{\mathtt{Y} = \mathtt{X}\}$
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 $\mathtt{X}$ 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) $\vdash\; \{P\}\; V\mathtt{:=}E\; \{P\mathtt{[}V/E\mathtt{]}\}$.

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

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

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

This has the clearly false consequence $\vdash\; \{\mathtt{X=0}\}\; \mathtt{X:=1}\; \{\mathtt{1=0}\}$ 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 go into this topic later in Chapter 4 where we give a *formal semantics* of our little programming language and then to *prove* that the axioms and rules of inference of 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 [17] showed that it failed to hold in six different ways for the (now obsolete) language Algol 60.

There is a 'forwards' version of the assignment axioms which is sometimes called Floyd's assignment axiom because it corresponds to the original semantics of assignment due to Floyd [9]. In this rule below, the existentially quantified variable v is the value of V in the state before executing the assignment (the initial state). The postcondition asserts that after the assignment, the value of V is the value of E evaluated in the initial state (hence $E\texttt{[}v/V\texttt{]}$) and the precondition evaluated in the initial state (hence $P\texttt{[}v/V\texttt{]}$) continues to hold.

The Floyd assignment axiom

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

Where v is a new variable (i.e. doesn't equal V or occur in P or E)

An example instance is:

$$\vdash\ \{\texttt{X=1}\}\ \texttt{X:=X+1}\ \{\exists v.\ \texttt{X} = \texttt{X+1[}v\texttt{/X]}\ \wedge\ \texttt{X=1[}v\texttt{/X]}\}$$

Simplifying the postcondition of this:

$$\vdash\ \{\texttt{X=1}\}\ \texttt{X:=X+1}\ \{\exists v.\ \texttt{X} = \texttt{X+1[}v\texttt{/X]}\ \wedge\ \texttt{X=1[}v\texttt{/X]}\}$$

$$\vdash\ \{\texttt{X=1}\}\ \texttt{X:=X+1}\ \{\exists v.\ \texttt{X} = v + \texttt{1}\ \wedge\ v = \texttt{1}\}$$

$$\vdash\ \{\texttt{X=1}\}\ \texttt{X:=X+1}\ \{\exists v.\ \texttt{X} = \texttt{1} + \texttt{1}\ \wedge\ v = \texttt{1}\}$$

$$\vdash\ \{\texttt{X=1}\}\ \texttt{X:=X+1}\ \{\texttt{X} = \texttt{1} + \texttt{1}\ \wedge\ \exists v.\ v = \texttt{1}\}$$

$$\vdash\ \{\texttt{X=1}\}\ \texttt{X:=X+1}\ \{\texttt{X} = \texttt{2}\ \wedge\ \texttt{T}\}$$

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

The Floyd assignment axiom is equivalent to standard one but harder to use because of the existential quantifier that it introduces. However, it is an important part of separation logic.

The Hoare assignment axiom is related to weakest preconditions (see Section 4.3.3) and the Floyd assignment axiom to strongest postconditions (see Section 4.4.1). As will be explained in the sections mentioned in the previous sentence:

Hoare assignment axiom: $\{\texttt{wlp(}V\texttt{:=}E\texttt{,}Q\texttt{)}\}\ V\texttt{:=}E\ \{Q\}$

Floyd assignment axiom: $\{P\}\ V\texttt{:=}E\ \{\texttt{sp(}V\texttt{:=}E\texttt{,}P\texttt{)}\}$

where $\texttt{wlp(}C\texttt{,}Q\texttt{)}$ and $\texttt{sp(}C\texttt{,}P\texttt{)}$ denote the weakest liberal precondition and strongest postcondition, respectively (see sections 4.3.3 and 4.4.1).

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 expressions of the form $\texttt{(}C\texttt{;}E\texttt{)}$, where C is a command and E an expression. Such an expression is evaluated by first executing C and then evaluating E and returning the resulting value as the value of $\texttt{(}C\texttt{;}E\texttt{)}$. Thus the evaluation of the expression may cause a 'side effect' resulting from the execution of C. For example `(Y:=1; 2)` 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 $\texttt{(}C\texttt{;}E\texttt{)}$, then it could be used to deduce:

$\vdash\ \{\texttt{Y=0}\}\ \texttt{X:=(Y:=1; 2)}\ \{\texttt{Y=0}\}$

(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`.

2.1.2 Precondition strengthening

The next rule of Hoare logic enables the preconditions of (i) and (ii) on page 18 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 18 it follows by precondition strengthening that

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

The variable `n` is an example of an *auxiliary* (or *ghost*) variable. As described earlier (see page 10), 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 `X` is `n`, then after executing the assignment `X:=X+1` its value will be `n+1`.

2. From the logical truth $\vdash$ `T` $\Rightarrow$ `(`E`=`E`)`, and 3 on page 18 one can deduce that if `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 $\vdash\ \{P\}\ C\ \{Q_1 \wedge Q_2\}$ to be proved by proving separately that both $\vdash\ \{P\}\ C\ \{Q_1\}$ and $\vdash\ \{P\}\ C\ \{Q_2\}$.

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 conditional rule

The conditional rule

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

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 rule (and others) it follows that

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

2.1.8 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 earlier rules:

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

Hence by precondition strengthening

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

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

$$\begin{array}{ll}\text{(i)} & \vdash \{\mathtt{X{=}R{+}(Y{\times}Q)}\}\\ & \quad \mathtt{WHILE\ Y{\leq}R\ DO\ (R{:=}R{-}Y;\ Q{:=}Q{+}1)}\\ & \ \{\mathtt{X{=}R{+}(Y{\times}Q)}{\wedge}\neg\mathtt{(Y{\leq}R)}\}\end{array}$$

By applying the assignment axiom twice, it is easy to deduce that

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

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

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

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:

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

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 $\mathtt{T} \wedge \neg\mathtt{T}$ could not hold in the resulting state.

2.1.9 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 **??**). An early reference on the logic of FOR-commands is Hoare's 1972 paper [14]; a comprehensive treatment can be found in Reynolds [**?**].

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

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

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 VAR V;
  V:=e1; C; V:=e1+1; C ; ... ; V:=e2; C
END
```

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 21,

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

Hence by the sequencing rule

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

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

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

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

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

Hence by the derived sequencing rule:

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

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

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

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

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

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

hence the rule allows us the following to be derived:

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

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

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

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

$$\begin{array}{l}\vdash \{\texttt{(X=((1-1)}\times\texttt{1) DIV 2)}\wedge\texttt{(1}\leq\texttt{M)}\}\\ \quad\texttt{FOR N:=1 UNTIL M DO X:=X+N}\\ \quad\quad\{\texttt{X=(M}\times\texttt{(M+1)) DIV 2}\}\end{array}$$

Hence

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

Note that if

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

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

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

then by precondition strengthening one can infer

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

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

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

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

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

Tennent's FOR-rule

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

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.

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.

2.1.10 Arrays

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

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

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

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

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

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

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

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 ⊢ `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:

$$\vdash \{\texttt{A(Y)=y} \wedge \texttt{A(X)=x}\}$$
```
R := A(X); A(X) := A(Y); A(Y) := R
```
$$\{\texttt{A(X)=y} \wedge \texttt{A(Y)=x}\}$$

The desired result follows from the block rule.

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

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

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

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

This is not enough, however, because SORTED(A,N) can be achieved by simply zeroing the first N elements of A.
It is necessary to require that the sorted array is a rearrangement, or permutation, of the original array.

To formalize this, let $\texttt{PERM(}A, A', N\texttt{)}$ 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

$$\vdash \{\texttt{1}\leq\texttt{N} \wedge \texttt{A=a}\}$$
```
N:=1
```
$$\{\texttt{SORTED(A,N)} \wedge \texttt{PERM(A,a,N)}\}$$

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

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

Chapter 3

Mechanizing Program Verification

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

After doing only a few examples, 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 $\vdash$ $(\mathtt{R}{=}\mathtt{X} \wedge \mathtt{Q}{=}0) \Rightarrow (\mathtt{X} = \mathtt{R} + \mathtt{Y}{\times}\mathtt{Q})$).

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 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 [10]. 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:

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

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

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

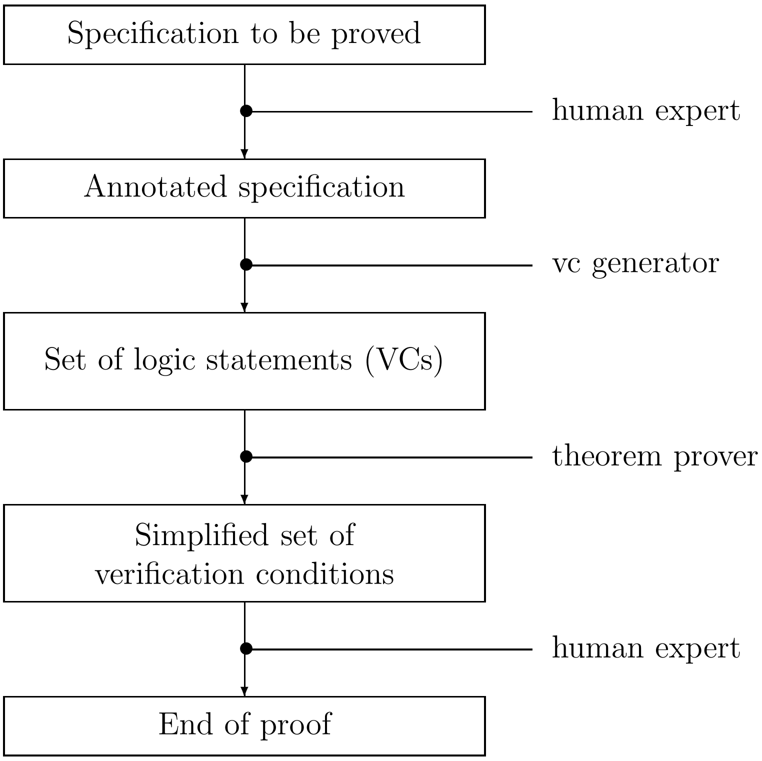
(because $\vdash \{\texttt{R=x} \wedge \texttt{X=y}\}\ \texttt{Y:=R}\ \{\texttt{Y=x} \wedge \texttt{X=y}\}$). By a similar argument this subgoal can be reduced to

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

which clearly follows from the assignment axiom.

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

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.


Specification to be proved
human expert
Annotated specification
vc generator
Set of logic statements (VCs)
theorem prover
Simplified set of verification conditions
human expert
End of proof

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 VCs). 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 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 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 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 formalised. 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* (VCs 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 24)

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

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

```
{T}
   R:=X;
   Q:=0; {R=X ∧ Q=0} ←—P₁
   WHILE Y≤R DO {X = R+Y×Q} ←—P₂
     (R:=R-Y; Q:=Q+1)
{X = R+Y×Q ∧ R<Y}
```

The annotations $\mathtt{P}_1$ and $\mathtt{P}_2$ state conditions which are intended to hold *whenever* control reaches them. Control only reaches the point at which $\mathtt{P}_1$ is placed once, but it reaches $\mathtt{P}_2$ each time the `WHILE` body is executed and whenever this happens $\mathtt{P}_2$ (i.e. `X=R+Y×Q`) holds, even though the values of `R` and `Q` vary. $\mathtt{P}_2$ 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` 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}
   Y:=1; ⟵①
   WHILE X≠0 DO ⟵②
     (Y:=Y×X; X:=X-1)
{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

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

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).

Conditionals

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

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

If $C_1; \ldots ; C_n$ is properly annotated, then (see page 39) 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

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

The only verification condition generated by this is

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

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.

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 24, 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

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

are:

(i) `R=X ∧ Q=0 ⇒ (X = R+(Y×Q))`

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

together with the verification condition for

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

which consists of the single condition

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

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

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

if

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

and

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

and

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

3.5 Justification of verification conditions

It will be shown in this section that an annotated specification $\{P\}C\{Q\}$ is provable in 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 [8].

It is easy to show 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. For example, the verification conditions from the annotated specification $\{\mathtt{T}\}$ `WHILE F DO` $\{\mathtt{F}\}$ `X:=0` $\{\mathtt{T}\}$ are not provable, but this Hoare triple is provable in Hoare logic.

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 $P \Rightarrow Q\mathtt{[}E/V\mathtt{]}$. If this is provable, then as $\vdash \{Q\mathtt{[}E/V\mathtt{]}\}V{:=}E\{Q\}$ (by the assignment axiom on page 18) it follows by precondition strengthening (page 20) that $\vdash \{P\}V := E\{Q\}$.

Conditionals

If the verification conditions for $\{P\}$ `IF` S `THEN` C_1 `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 conditional rule (page 24) $\vdash \ \{P\}$ `IF` S `THEN` C_1 `ELSE` C_2 $\{Q\}$.

Sequences

There are two cases to consider:

(i) If the verification conditions for $\{P\}\ C_1;\ldots;C_{n-1};\{R\}C_n\ \{Q\}$ are provable, then the verification conditions for $\{P\}\ C_1;\ldots;C_{n-1}\ \{R\}$ and $\{R\}\ C_n\ \{Q\}$ must both be provable and hence by induction we have $\vdash \ \{P\}\ C_1;\ldots;C_{n-1}\ \{R\}$ and $\vdash \ \{R\}\ C_n\ \{Q\}$. Hence by the sequencing rule (page 22) $\vdash \ \{P\}\ C_1;\ldots;\ C_{n-1};C_n\ \{Q\}$.

(ii) If the verification conditions for $\{P\}\ C_1;\ldots;C_{n-1};V := E\ \{Q\}$ are provable, then it must be the case that the verification conditions for

$\{P\}\ C_1;\ \ldots;C_{n-1}\ \{Q\texttt{[}E/V\texttt{]}\}$ are also provable and hence by induction we have $\vdash\ \{P\}\ C_1;\ \ldots;C_{n-1}\ \{Q\texttt{[}E/V\texttt{]}\}$. It then follows by the assignment axiom that $\vdash\ \{Q\texttt{[}E/V\texttt{]}\}\ V := E\ \{Q\}$, hence by the sequencing rule $\vdash\ \{P\}\ C_1;\ \ldots;C_{n-1};V := E\{Q\}$.

`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 24) $\vdash\ \{R\}$ `WHILE` S `DO` $C\ \{R\ \wedge\ \neg S\}$, hence by the consequence rules (see page 22) $\vdash\ \{P\}$ `WHILE` S `DO` $C\ \{Q\}$.

Chapter 4

Soundness and Completeness

The question of whether the axioms and rules of Hoare logic are correct (soundness) and sufficient (completeness) is investigated. This requires the meaning (semantics) of the programming language to be formulated explicitly so that the semantics of Hoare triples can be rigorously defined.

4.1 Semantics

A command C transforms an initial state into a final state (or fails to terminate). For the language described so far there is at most one final state reachable from a given initial state – i.e. commands are deterministic – but this will not be the case later, when we add storage allocation to our language. There are several essentially equivalent ways to represent the meaning of commands mathematically. We will use relations, but partial functions are often used. Use of relations is associated with operational semantics and partial functions with denotational semantics, however this is not rigid: denotational semantics can use relations as denotations and operational semantics can inductively define functions. In fact, in Section 4.1.2 below, we give a denotational semantics of commands in which the denotations are relations.

The various styles of semantics are largely just different ways of representing the same mathematical ideas. Some mathematical representations are better suited for some purposes and other representations for others. The semantics I give below may or may not correspond to semantics you have seen before in earlier courses. If it seems different, then a good exercise is to think about how it is related.

We are going to represent the meaning of a command C by a binary relation on the set of states: s_1 is related to s_2 in this relation iff when C is executed in state s_1 it terminates in state s_2.

There are several ways of representing relations mathematically and although it doesn't really matter which one is chosen, it may help avoid confusion in what follows if we say a little about these alternative representations here, before diving into specific details.

Introductory books on set theory usually represent relations as sets of ordered pairs, so x is related to y by relation R iff $(x, y) \in R$. Thus a binary relation R between sets X and Y is a subset of $X \times Y$, i.e. $R \subseteq (X \times Y)$ or, equivalently, $R \in \mathcal{P}(X \times Y)$, where $\mathcal{P}$ is the powerset operator. If S is any set then any subset $A \subseteq S$ can be characterised by a function $f_A : S \to \{\mathsf{T}, \mathsf{F}\}$ defined by:

$$\forall s \in S.\ f_A(s) = \mathsf{T} \Leftrightarrow s \in A$$

f_A is called the *characteristic function* of A. Thus a relation $R \subseteq (X \times Y)$ can be characterised by its characteristic function f_R defined by:

$$\forall x \in X.\ \forall y \in Y.\ f_R(x, y) = \mathsf{T} \ \Leftrightarrow\ (x, y) \in R$$

where $f_R : (X \times Y) \to \{\mathsf{T}, \mathsf{F}\}$. If the set of functions from set S to set T is denoted by $(S \to T)$ and *Bool* is the set $\{\mathsf{T}, \mathsf{F}\}$, then $f_R \in ((X \times Y) \to \mathit{Bool})$. You may recall from earlier courses (e.g. on ML) that functions that take two or more arguments can be 'curried' so that they take the arguments one at a time. If we curry f_R we get a function f_R^{curried} defined by:

$$f_R^{\mathit{curried}}\ x\ y = f_R(x, y)$$

and then $f_R^{\mathit{curried}} : X \to (Y \to \mathit{Bool})$ or $f_R^{\mathit{curried}} : X \to Y \to \mathit{Bool}$ if we assume the standard convention that → associates to the right. Note that we also have $f_R^{\mathit{curried}} \in \mathcal{P}(X \to Y \to \mathit{Bool})$.

To sum up, a relation R can be represented by a set of pairs, by a characteristic function that maps pairs to Booleans, or by the curried characteristic function. For a somewhat arbitrary mixture of historical and stylistic reasons, we are going to use the curried characteristic function representation of relations to represent the semantics of commands. Specifically, we are going to define $\mathsf{Csem}\ C\ s_1\ s_2$ to mean that if command C is started in state s_1 then it can terminate in state s_2. Here $\mathsf{Csem}\ C$ is the relation that represents the semantics of C, represented as a curried characteristic function. The set of commands in our language will be denoted by *Com* and the set of states will be denoted by *State*. Thus $\mathsf{Csem}\ C : \mathit{State} \to \mathit{State} \to \mathit{Bool}$ or $\mathsf{Csem} : \mathit{Com} \to \mathit{State} \to \mathit{State} \to \mathit{Bool}$. As mentioned earlier, the choice of representing $\mathsf{Csem}\ C$ as a curried characteristic function, rather than as a set of ordered pairs, is more a matter of style than substance.

Let *Var* be the set of variables that are allowed in statements, expressions and commands and *Val* be the set of values that variables can take. It is not necessary to be specific about what variables and values actually are: *Var* could be, for example, the set of finite strings of ASCII characters and *Val* could be the set of integers. A state determines the value of each variable and, in addition, may contain other information. For our little programming language it is sufficient to take the state to be a function from *Var* to *Val*. Using the notation $A \to B$ to denote the set of functions with domain A and range (codomain) B we define the set *State* of states by:

$$\mathit{State} = Var \to \mathit{Val}$$

Note that the following are all equivalent $s \in \mathit{State}$, $s \in (\mathit{Var} \to \mathit{Val})$ and $s : \mathit{Var} \to \mathit{Val}$. I will sometimes use $s(v)$ and sometimes $s\ v$ for the value associated with variable v in state s (i.e. the application of the function representing the state s to v). Although in this chapter it is sufficient to represent states as functions from variables to values, in Chapter 7 we will need to add another components to the state to represents the contents of pointers. We will extend the definition of *State* in that chapter. Particular states can be defined using λ-notation. For example, the state that maps `X` to `1`, `Y` to `2` and everything else to `0` is defined by:

$$\lambda v.\ \mathit{if}\, v{=}\mathtt{X}\ \mathit{then}\, \mathtt{1}\ \mathit{else}\, (\mathit{if}\, v{=}\mathtt{Y}\ \mathit{then}\, \mathtt{2}\ \mathit{else}\, \mathtt{0})$$

If $s \in \mathit{State}$, $v \in \mathit{Var}$ and $n \in \mathit{Val}$ then $s\mathtt{[}n\mathtt{/}v\mathtt{]}$ denotes that state that is the same as s, except for the value of variable v is 'updated' to be n. Thus $s\mathtt{[}n\mathtt{/}v\mathtt{]}$ is given by the equation:

$$s\mathtt{[}n\mathtt{/}v\mathtt{]} \ = \ \lambda v'.\ \mathit{if}\, v' = v\ \mathit{then}\, n\ \mathit{else}\, s(v') \qquad \text{(where } v' \text{ is a new variable)}$$

Example:

$$(\lambda v.\ \mathit{if}\, v{=}\mathtt{X}\ \mathit{then}\, \mathtt{1}\ \mathit{else}\, (\mathit{if}\, v{=}\mathtt{Y}\ \mathit{then}\, \mathtt{2}\ \mathit{else}\, \mathtt{0}))\mathtt{[3/Z]} =$$
$$\lambda v.\ \mathit{if}\, v{=}\mathtt{X}\ \mathit{then}\, \mathtt{1}\ \mathit{else}\, (\mathit{if}\, v{=}\mathtt{Y}\ \mathit{then}\, \mathtt{2}\ \mathit{else}\, (\mathit{if}\, v{=}\mathtt{Z}\ \mathit{then}\, \mathtt{3}\ \mathit{else}\, \mathtt{0}))$$

4.1.1 Semantics of expressions and statements

Commands may contain expressions or statements: expressions occur on the right hand side of assignments and statements occur in the tests of conditionals and `WHILE`-commands. The precondition and postcondition of Hoare triples are also statements. The classical treatment of Hoare logic was built upon first order logic, expressions were taken to be terms of logic and statements to be formulae. You will be familiar with the semantics of first order logic from earlier courses and I do not want to repeat that material here. Furthermore, in modern applications, the language used for writing preconditions and postconditions is now sometimes weaker or stronger than first order logic, e.g. quantifier free logic (weaker) or higher order logic (stronger).

To avoid the details of particular logics and their semantics we will assume that we are given sets *Exp* and *Sta* of expressions and statements, together with semantic functions Esem and Ssem defining their semantics, where:

$$\begin{aligned}\mathsf{Esem} &: \mathit{Exp} \to \mathit{State} \to \mathit{Val}\\ \mathsf{Ssem} &: \mathit{Sta} \to \mathit{State} \to \mathit{Bool}\end{aligned}$$

We now give some informal discussion and examples to illustrate how Esem and Ssem might be defined for particular logics (i.e. for particular *Exp* and *Sta*). We hope it will be clear from this how a more formal treatment would go. In the usual logic terminology (e.g. as used in the IB Tripos course *Logic and Proof*) we are using states to represent interpretation functions and *Val* as the domain or universe. Variables are interpreted by looking them up in the state:

$$\mathsf{Esem}\ \mathtt{X}\ s = s(\mathtt{X})$$

Constants get their usual mathematical or logical meaning:

$$\begin{aligned}&\mathsf{Esem}\ \mathtt{3}\ s = 3\\ &\mathsf{Ssem}\ \mathtt{T}\ s = \mathsf{T}\end{aligned}$$

Compound expressions or statements are interpreted bottom up: the (recursively computed) value of sub-expressions is combined using appropriate mathematical or logical operators to get the interpretation of the whole expression. For example:

$$\begin{aligned}&\mathsf{Esem}\ (-E)\ s = -(\mathsf{Esem}\ E\ s)\\ &\mathsf{Esem}\ (E_1 + E_2)\ s = (\mathsf{Esem}\ E_1\ s) + (\mathsf{Esem}\ E_2\ s)\end{aligned}$$

$$\begin{aligned}&\mathsf{Ssem}\ (\neg S)\ s = \neg(\mathsf{Ssem}\ S\ s)\\ &\mathsf{Ssem}\ (S_1 < S_2)\ s = (\mathsf{Ssem}\ S_1\ s) < (\mathsf{Ssem}\ S_2\ s)\end{aligned}$$

where the symbols "$-$", "$+$", "$\neg$" and "$<$" on the left hand side of these equations are part of the syntax of statements (i.e. part of the object language) and those on the right hand side are informal mathematical notation (i.e. part of our metalanguage). This is a subtle point worth pondering! Quantifiers (which may occur in preconditions and postconditions, but probably not in tests in commands) are interpreted in the standard way:

$$\begin{array}{l}\mathsf{Ssem}\ (\forall v.\ S)\ s = \forall n\ \mathsf{Ssem}\ S\ (s\,\texttt{[}n\texttt{/}v\texttt{]})\\ \mathsf{Ssem}\ (\exists v.\ S)\ s = \exists n.\ \mathsf{Ssem}\ S\ (s\,\texttt{[}n\texttt{/}v\texttt{]})\end{array}$$

Example:

$$\mathsf{Ssem}\ (\texttt{Y<Z+3})\ (\lambda v.\ \mathit{if}\, v{=}\texttt{X}\ \mathit{then}\,\texttt{1}\ \mathit{else}\,(\mathit{if}\, v{=}\texttt{Y}\ \mathit{then}\,\texttt{2}\ \mathit{else}\,\texttt{0})) = (2{<}0{+}3)$$

I hope this is sufficient explanation of Esem and Ssem for what follows. Note that for any $E \in \mathit{Exp}$ and $S \in \mathit{Sta}$ it is the case that:

$$\begin{array}{lcl}\mathsf{Esem}\ E & : & \mathit{State} \to \mathit{Val}\\ \mathsf{Ssem}\ S & : & \mathit{State} \to \mathit{Bool}\end{array}$$

4.1.2 Semantics of commands

Csem $C\ s_1\ s_2$ will be defined recursively bottom up. The only commands that don't contain sub-commands are assignments. After an assignment the final state s_2 is equal to the initial state with the variable V on the left hand side of the assignment updated to have the value of the expression E on the right hand side of the assignment in the initial state.

$$\boxed{\mathsf{Csem}\ (V\texttt{:=}E)\ s_1\ s_2\ =\ (s_2 = s_1\texttt{[}(\mathsf{Esem}\ E\ s_1)\texttt{/}V\texttt{]})}$$

A final state s_2 can be reached by executing a sequence $C_1\texttt{;}C_2$ starting in an initial state s_1 iff there is an intermediate state s reachable by executing C_1 in s_1 and s_2 is reachable from this intermediate state by executing C_2.

$$\boxed{\mathsf{Csem}\ (C_1\texttt{;}C_2)\ s_1\ s_2\ =\ \exists s.\ \mathsf{Csem}\ C_1\ s_1\ s\ \wedge\ \mathsf{Csem}\ C_2\ s\ s_2}$$

If S is true in a state s_1 then state s_2 can be reached by executing the conditional $\texttt{IF}\ S\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2$ starting in s_1 iff s_2 can be reached by executing the THEN-branch C_1 starting in s_1. However, if S is false in a state s_1 then state s_2 can be reached by executing conditional starting in s_1 iff s_2 can be reached by executing the ELSE-branch C_2 starting in s_1.

$$\boxed{\begin{array}{l}\mathsf{Csem}\ (\texttt{IF}\ S\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2)\ s_1\ s_2 =\\ \ \ \mathit{if}\ \mathsf{Ssem}\ S\ s_1\ \ \mathit{then}\ \mathsf{Csem}\ C_1\ s_1\ s_2\ \ \mathit{else}\ \mathsf{Csem}\ C_2\ s_1\ s_2\end{array}}$$

If final state s_2 can be reached from initial state s_1 by executing $\texttt{WHILE}\ S\ \texttt{DO}\ C$, then there must be some finite number of iterations of C that will reach s_2, S must be true in all the intermediate states and false in s_2. This is formalised by defining a function Iter that iterates a finite number of times and then defining:

$$\boxed{\mathsf{Csem}\ (\texttt{WHILE}\ S\ \texttt{DO}\ C)\ s_1\ s_2\ =\ \exists n.\ \mathsf{Iter}\ n\ (\mathsf{Ssem}\ S)\ (\mathsf{Csem}\ C)\ s_1\ s_2}$$

The function Iter is defined by recursion on n as follows:

$$\begin{array}{ll}\mathsf{Iter}\ 0\ p\ c\ s_1\ s_2 & = \neg(p\ s_1) \wedge (s_1{=}s_2) \\ \mathsf{Iter}\ (n{+}1)\ p\ c\ s_1\ s_2 & = p\ s_1 \wedge \exists s.\ c\ s_1\ s \wedge \mathsf{Iter}\ n\ p\ c\ s\ s_2\end{array}$$

The first argument n of Iter is the number of iterations. The second argument p is a predicate on states (e.g. $\mathsf{Ssem}\ S$). The third argument c is a curried characteristic function (e.g. $\mathsf{Csem}\ C$). The fourth and fifth arguments are the initial and final states, respectively. If *Num* is the set of natural numbers $\{0, 1, 2, \ldots\}$, then:

$$\mathsf{Iter} : \mathit{Num}{\to}(\mathit{State}{\to}\mathit{Bool}){\to}(\mathit{State}{\to}\mathit{State}{\to}\mathit{Bool}){\to}\mathit{State}{\to}\mathit{State}{\to}\mathit{Bool}$$

4.2 Soundness of Hoare logic

The meaning of a Hoare triple $\{P\}\ C\ \{Q\}$ is defined to be $\mathsf{Hsem}\ P\ C\ Q$ where:

$$\boxed{\mathsf{Hsem}\ P\ C\ Q\ =\ \forall s_1\ s_2.\ \mathsf{Ssem}\ P\ s_1 \wedge \mathsf{Csem}\ C\ s_1\ s_2 \Rightarrow \mathsf{Ssem}\ Q\ s_2}$$

This definition can be used to formulate the soundness of Hoare logic. To do this we must prove that all instances of the assignment axiom are true, and that all conclusions deduced using inference rules are true if the hypotheses are true. Recall the assignment axiom:

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.

To prove this sound we must show that for every V, E and P:

$$\mathsf{Hsem}\ (P\texttt{[}E/V\texttt{]})\ (V\texttt{:=}E)\ P$$

Unfolding the definition of Hsem converts this to:

$$\forall s_1\ s_2.\ \mathsf{Ssem}\ (P\texttt{[}E/V\texttt{]})\ s_1 \wedge \mathsf{Csem}\ (V\texttt{:=}E)\ s_1\ s_2 \Rightarrow \mathsf{Ssem}\ P\ s_2$$

Unfolding the definition of Csem converts this to:

$$\forall s_1\ s_2.\ \mathsf{Ssem}\ (P\texttt{[}E/V\texttt{]})\ s_1 \wedge (s_2 = s_1\texttt{[}(\mathsf{Esem}\ E\ s_1)\texttt{/}V\texttt{]}) \Rightarrow \mathsf{Ssem}\ P\ s_2$$

which simplifies to:

$$\forall s_1.\ \mathsf{Ssem}\ (P\texttt{[}E/V\texttt{]})\ s_1 \Rightarrow \mathsf{Ssem}\ P\ (s_1\texttt{[}(\mathsf{Esem}\ E\ s_1)\texttt{/}V\texttt{]})$$

This may appear confusing since it uses the notation $\texttt{[}\cdots\texttt{/}\cdots\texttt{]}$ with different meanings in the antecedent (the left argument of $\Rightarrow$) and consequent (the right argument). In the antecedent, $P\texttt{[}E/V\texttt{]}$ denotes the result of substituting the expression E for the variable V in the statement P. In the consequent $s_1\texttt{[}(\mathsf{Esem}\ E\ s_1)\texttt{/}V\texttt{]}$ denotes the state obtained by updating s_1 so that the value of V is the value of E in s_1 (and the values of all other variables are unchanged).

Diversion on substitution.

We have avoided specifying in detail exactly what the syntax of expressions and statements is, so it is not possible to prove general properties about them. However, for any reasonable definitions we would expect that:

$$\mathsf{Ssem}\ (P\texttt{[}E/V\texttt{]})\ s\ =\ \mathsf{Ssem}\ P\ (s\texttt{[}(\mathsf{Esem}\ E\ s)\texttt{/}V\texttt{]})$$

For example, take P to be `X+Y>Z`, E to be `X+1` and V to be `Y`, then the equation above becomes:

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

Now $\mathsf{Esem}\ (\texttt{X+1})\ s = s(\texttt{X}){+}1$ so the equation above becomes:

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

Evaluating the substitution on the left hand side reduces this to:

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

Evaluating the Ssem gives:

$$\begin{array}{l}(s(\texttt{X}){+}(s(\texttt{X}){+}1){>}s(\texttt{Z}))\ = \\ \qquad ((s\texttt{[}(s(\texttt{X}){+}1)\texttt{/Y]})(\texttt{X}){+}(s\texttt{[}(s(\texttt{X}){+}1)\texttt{/Y]})(\texttt{Y}){>}(s\texttt{[}(s(\texttt{X}){+}1)\texttt{/Y]})(\texttt{Z}))\end{array}$$

Using the definition of $s\,\texttt{[}n\texttt{/}v\texttt{]}$, and assuming $\texttt{X}$, $\texttt{Y}$ and $\texttt{Z}$ are distinct, enables the right hand side of this equation to be simplified, to give:

$$(s(\texttt{X})+(s(\texttt{X})+1)>s(\texttt{Z})) \;=\; (s(\texttt{X})+(s(\texttt{X})+1)>s(\texttt{Z}))$$

This is clearly true as the left and right hand sides are identical.

Although this is just an example, it illustrates why for all S, E, V and s it is the case that: $\mathsf{Ssem}\ (S\texttt{[}E\texttt{/}V\texttt{]})\ s \;=\; \mathsf{Ssem}\ S\ (s\texttt{[}(\mathsf{Esem}\ E\ s)\texttt{/}V\texttt{]})$

In fact, if this equation did not hold then one would have a bad definition of substitution – indeed this equation can be taken as the semantic specification of substitution!

End of diversion on substitution.

Returning to the soundness of the assignment axiom, recall that it was equivalent to the following holding for all P, E and V:

$$\forall s_1.\ \mathsf{Ssem}\ (P\texttt{[}E\texttt{/}V\texttt{]})\ s_1 \Rightarrow \mathsf{Ssem}\ P\ (s_1\texttt{[}(\mathsf{Esem}\ E\ s_1)\texttt{/}V\texttt{]})$$

If the equation for substitution motivated in the diversion above holds, then this implication holds too, since for any statements P and Q, if $P = Q$ then it follows that $P \Rightarrow Q$.

Thus, assuming the semantic substitution equation discussed above, we have shown that the assignment axiom is sound.

The soundness of the Hoare logic rules of inference is almost trivial except for the $\texttt{WHILE}$-rule, and even that is fairly straightforward. We will restate the rules and then outline the proof of their soundness.

Precondition strengthening

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

This rule is sound if the following is true for all P, P', C and Q:

$$(\forall s.\ \mathsf{Ssem}\ P\ s \Rightarrow \mathsf{Ssem}\ P'\ s) \land \mathsf{Hsem}\ P'\ C\ Q \Rightarrow \mathsf{Hsem}\ P\ C\ Q$$

which, after expanding the definition of Hsem, becomes:

$$\begin{array}{l}(\forall s.\ \mathsf{Ssem}\ P\ s \Rightarrow \mathsf{Ssem}\ P'\ s) \land \\ (\forall s_1\ s_2.\ \mathsf{Ssem}\ P'\ s_1 \land \mathsf{Csem}\ C\ s_1\ s_2 \Rightarrow \mathsf{Ssem}\ Q\ s_2) \\ \Rightarrow \\ \forall s_1\ s_2.\ \mathsf{Ssem}\ P\ s_1 \land \mathsf{Csem}\ C\ s_1\ s_2 \Rightarrow \mathsf{Ssem}\ Q\ s_2\end{array}$$

This is an instance of the statement below if we take p, p', q, c to be $\mathsf{Ssem}\ P$, $\mathsf{Ssem}\ P'$, $\mathsf{Ssem}\ Q$, $\mathsf{Csem}\ C$, respectively.

$$\begin{array}{l}(\forall s.\ p\ s \Rightarrow p'\ s)\ \wedge\ (\forall s_1\ s_2.\ p'\ s_1 \wedge c\ s_1\ s_2 \Rightarrow q\ s_2)\\ \Rightarrow\\ \forall s_1\ s_2.\ p\ s_1 \wedge c\ s_1\ s_2 \Rightarrow q\ s_2\end{array}$$

This is clearly true.

Postcondition weakening

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

This is sound by a similar argument.

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

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

This is sound by a similar argument.

The sequencing rule

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

This rule is sound if the following is true for all P, Q, R, C_1 and C_2:

$$\mathsf{Hsem}\ P\ C_1\ Q \wedge \mathsf{Hsem}\ Q\ C_2\ R \Rightarrow \mathsf{Hsem}\ P\ (C_1;C_2)\ R$$

which, after expanding the definition of Hsem, becomes:

$$\begin{array}{l}(\forall s_1\ s_2.\ \mathsf{Ssem}\ P\ s_1 \wedge \mathsf{Csem}\ C\ s_1\ s_2 \Rightarrow \mathsf{Ssem}\ Q\ s_2) \wedge \\ (\forall s_1\ s_2.\ \mathsf{Ssem}\ Q\ s_1 \wedge \mathsf{Csem}\ C\ s_1\ s_2 \Rightarrow \mathsf{Ssem}\ R\ s_2) \\ \Rightarrow \\ \forall s_1\ s_2.\ \mathsf{Ssem}\ P\ s_1 \wedge \mathsf{Csem}\ (C_1;C_2)\ s_1\ s_2 \Rightarrow \mathsf{Ssem}\ R\ s_2\end{array}$$

This is an instance of the statement below if we unfold the definition of $\mathsf{Csem}\ (C_1;C_2)$ and take p, q, r, c_1, c_2 to be $\mathsf{Ssem}\ P$, $\mathsf{Ssem}\ Q$, $\mathsf{Ssem}\ R$, $\mathsf{Csem}\ C_1$, $\mathsf{Csem}\ C_2$, respectively.

$$\begin{array}{l}(\forall s_1\ s_2.\ p\ s_1 \wedge c_1\ s_1\ s_2 \Rightarrow q\ s_2)\ \wedge\ (\forall s_1\ s_2.\ q\ s_1 \wedge c_2\ s_1\ s_2 \Rightarrow r\ s_2) \\ \Rightarrow \\ \forall s_1\ s_2.\ p\ s_1 \wedge (\exists s.\ c_1\ s_1\ s \wedge c_2\ s\ s_2) \Rightarrow r\ s_2\end{array}$$

This is clearly true.

The conditional rule

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

A similar argument to the one for the sequencing rule shows the conditional rule to be sound.

The `WHILE`-rule

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

This rule is sound if the following is true for all P, S and C:

$$\mathsf{Hsem}\ (P \wedge S)\ C\ P \Rightarrow \mathsf{Hsem}\ P\ (\texttt{WHILE}\ S\ \texttt{DO}\ C)\ (P \wedge \neg S))$$

which, after expanding the definition of Hsem, becomes:

$$\begin{array}{l}(\forall s_1\ s_2.\ \mathsf{Ssem}\ (P \wedge S)\ s_1 \wedge \mathsf{Csem}\ C\ s_1\ s_2 \Rightarrow \mathsf{Ssem}\ P\ s_2) \\ \Rightarrow \\ \forall s_1\ s_2.\ \mathsf{Ssem}\ P\ s_1 \wedge \mathsf{Csem}\ (\texttt{WHILE}\ S\ \texttt{DO}\ C)\ s_1\ s_2 \Rightarrow \mathsf{Ssem}\ (P \wedge \neg S)\ s_2\end{array}$$

Using the equations $\mathsf{Ssem}\ (P \wedge Q)\ s_1 = \mathsf{Ssem}\ P\ s_1 \wedge \mathsf{Ssem}\ Q\ s_1$ and $\mathsf{Ssem}\ (P \wedge \neg Q)\ s_2 = \mathsf{Ssem}\ P\ s_2 \wedge \neg(\mathsf{Ssem}\ Q\ s_1)$ and expanding the definition of $\mathsf{Hsem}\ (\texttt{WHILE}\ S\ \texttt{DO}\ C)$ converts this to:

$$\begin{array}{l}(\forall s_1\ s_2.\ \mathsf{Ssem}\ P\ s_1 \wedge \mathsf{Ssem}\ S\ s_1 \wedge \mathsf{Csem}\ C\ s_1\ s_2 \Rightarrow \mathsf{Ssem}\ P\ s_1\\ \Rightarrow\\ \forall s_1\ s_2.\ \mathsf{Ssem}\ P\ s_1 \wedge (\exists n.\ \mathsf{Iter}\ n\ (\mathsf{Ssem}\ S)\ (\mathsf{Csem}\ C)\ s_1\ s_2)\\ \qquad\quad \Rightarrow\\ \qquad\quad \mathsf{Ssem}\ P\ s_2 \wedge \neg(\mathsf{Ssem}\ S\ s_2)\end{array}$$

This is an instance of the statement below if we take p, b, c to be $\mathsf{Ssem}\ P$, $\mathsf{Ssem}\ S$, $\mathsf{Csem}\ C$, respectively.

$$\begin{array}{l}(\forall s_1\ s_2.\ p\ s_1 \wedge b\ s_1 \wedge c\ s_1\ s_2 \Rightarrow p\ s_1)\\ \Rightarrow\\ \forall s_1\ s_2.\ p\ s_1 \wedge (\exists n.\ \mathsf{Iter}\ n\ b\ c\ s_1\ s_2) \Rightarrow p\ s_2 \wedge \neg(b\ s_2)\end{array}$$

which is equivalent to:

$$\begin{array}{l}(\forall s_1\ s_2.\ p\ s_1 \wedge b\ s_1 \wedge c\ s_1\ s_2 \Rightarrow p\ s_1)\\ \Rightarrow\\ \forall n\ s_1\ s_2.\ p\ s_1 \wedge \mathsf{Iter}\ n\ b\ c\ s_1\ s_2 \Rightarrow p\ s_2 \wedge \neg(b\ s_2)\end{array}$$

To prove this, assume the antecedent and show the consequent by induction of n. The basis ($n = 0$ case) is clearly true as 'false implies everything'. For the induction step assume:

1. $\forall s_1\ s_2.\ p\ s_1 \wedge b\ s_1 \wedge c\ s_1\ s_2 \Rightarrow p\ s_1$ (Hoare rule hypothesis)
2. $\forall s_1\ s_2.\ p\ s_1 \wedge \mathsf{Iter}\ n\ b\ c\ s_1\ s_2 \Rightarrow p\ s_2 \wedge \neg(b\ s_2)$ (induction hypothesis)

From these we must show the induction conclusion:

$$p\ s_1 \wedge \mathsf{Iter}\ (n{+}1)\ b\ c\ s_1\ s_2 \Rightarrow p\ s_2 \wedge \neg(b\ s_2)$$

Using the recursive definition of $\mathsf{Iter}\ (n{+}1)$ converts this to:

$$p\ s_1 \wedge (b\ s_1 \wedge \exists s.\ c\ s_1\ s \wedge \mathsf{Iter}\ n\ b\ c\ s\ s_2) \Rightarrow p\ s_2 \wedge \neg(b\ s_2)$$

which with a bit of quantifier fiddling is equivalent to:

$$p\ s_1 \wedge b\ s_1 \wedge c\ s_1\ s \wedge \mathsf{Iter}\ n\ b\ c\ s\ s_2 \Rightarrow p\ s_2 \wedge \neg(b\ s_2)$$

Which follows from the Hoare rule hypothesis and the induction hypothesis (i.e. 1 and 2 above) by a bit of implication chaining.

4.3 Decidability and completeness

$\{\mathtt{T}\}C\{\mathtt{F}\}$ is true if and only if C does not terminate, therefore, since the halting problem is undecidable, so is Hoare logic.

Soundness is that any Hoare triple that can be deduced using the axioms and rules of inference of Hoare logic is true. The converse, completeness, would be that any true Hoare triple could be deduced using the axioms and rules of Hoare logic. Unfortunately, this cannot hold in general.

Consider $\{\mathtt{T}\}\ \mathtt{X:=X}\ \{P\}$. According to the semantics above, this is true iff $\mathsf{Hsem}\ \mathtt{T}\ (\mathtt{X:=X})\ P$ is true, i.e.:

$$\forall s_1\ s_2.\ \mathsf{Ssem}\ \mathtt{T}\ s_1 \wedge \mathsf{Csem}\ (\mathtt{X:=X})\ s_1\ s_2 \Rightarrow \mathsf{Ssem}\ P\ s_2$$

Since $\mathsf{Csem}\ (\mathtt{X:=X})\ s_1\ s_2 = (s_2 = s_1)$ and $\mathsf{Ssem}\ \mathtt{T}\ s_1 = \mathsf{T}$ this reduces to:

$$\forall s_1\ s_2.\ \mathsf{T} \wedge (s_2 = s_1) \Rightarrow\ \mathsf{Ssem}\ P\ s_2$$

which, by specialising s_1 and s_2 to s, simplifies to $\forall s.\ \mathsf{Ssem}\ P\ s$ – i.e. P is true. Thus if we could deduce any true Hoare triple using Hoare logic then we would be able to deduce any true statement of the specification language using Hoare logic! Most logics suitable for specifying programs are incomplete (e.g. first order arithmetic), so Hoare logic cannot be complete.

However the kind of completeness just described above is only impossible due to the incompleteness of the specification language used for preconditions and postconditions. If we separate the 'programming logic' from the 'specification logic', then it is possible to formulate a sort of completeness, called *relative completeness* [6], that provides some reassurance that Hoare logic is adequate for reasoning about the small collection of simple commands we have discussed – i.e. there are no 'missing' axioms or rules. It turns out, however, that even this limited kind of completeness may be impossible for constructs found in many real languages (but not in our 'toy' language) [5].

We will not attempt to explain the exact details of Cook's and others' work on relative completeness, as both the technical logical issues and also their intuitive interpretation are quite subtle [1, 16]. Furthermore doing this would require us to be more precise than we wish about the syntax, semantics and proof theory of the specification language in which preconditions and postconditions are expressed. We will, however, sketch the key ideas. A concept that is used in proving relative completeness is the *weakest precondition*. This concept is not only useful for its role in showing relative completeness, it also has practical applications, including providing an improved approach to verification conditions (which we discuss later) and as the foundation for theories of *program refinement*.

4.3.1 Relative completeness

We are going to explain the idea of relative completeness and also show that it holds for our little programming language by using *weakest liberal preconditions*. However, we will put off the detailed definition and analysis of these until Section 4.3.2. In this section we say just enough about what they are and what properties they have (P_1, P_2 and P_3 below) so that we can explain relative completeness.

For each command C and statement Q we assume there is a statement $\texttt{wlp(}C\texttt{,}Q\texttt{)}$ – intuitively the weakest precondition such that ensures Q holds after executing C – with the property that:

$$\vdash\ \{\texttt{wlp(}C\texttt{,}Q\texttt{)}\}\ C\ \{Q\} \qquad (\mathrm{P}_1)$$

The existence of the statement $\texttt{wlp(}C\texttt{,}Q\texttt{)}$ in the specification language depends on the specification language being strong enough. A language strong enough to enable $\texttt{wlp(}C\texttt{,}Q\texttt{)}$ to be defined is called *expressive*.

The operator `wlp` constructs a statement, which is a sentence in a formal language, from a command and another statement. Thus `wlp` constructs a syntactic thing (a statement) from other syntactic things (a command and a statement). We also assume a semantic counterpart to `wlp` called Wlp which operates on the meanings of commands (functions representing binary relations on states) and the meanings of statements (predicates on states). Wlp is a curried function:

$$\mathsf{Wlp} : (\mathit{State} \to \mathit{State} \to \mathit{Bool}) \to (\mathit{State} \to \mathit{Bool}) \to \mathit{State} \to \mathit{Bool}$$

we assume the following property connecting `wlp` and Wlp for all commands C and statements Q:

$$\mathsf{Ssem}\ (\texttt{wlp(}C\texttt{,}Q\texttt{)})\ =\ \mathsf{Wlp}\ (\mathsf{Csem}\ C)(\mathsf{Ssem}\ Q) \qquad (\mathrm{P}_2)$$

Notice that this is an equation between predicates. We also assume:

$$\mathsf{Hsem}\ P\ C\ Q\ =\ \forall s.\ \mathsf{Ssem}\ P\ s \Rightarrow \mathsf{Wlp}\ (\mathsf{Csem}\ C)\ (\mathsf{Ssem}\ Q)\ s \qquad (\mathrm{P}_3)$$

The shape of the relative completeness proof can now be sketched. Assume $\{P\}\ C\ \{Q\}$ is true, i.e. $\mathsf{Hsem}\ P\ C\ Q$ is true. We will show that the statement $P \Rightarrow \texttt{wlp(}C\texttt{,}Q\texttt{)}$ must also be true – assume this for now. If we could prove this true statement, i.e. had $\vdash\ P \Rightarrow \texttt{wlp(}C\texttt{,}Q\texttt{)}$, then by precondition strengthening and the property P_1 it would follow that $\vdash\ \{P\}\ C\ \{Q\}$ by Hoare logic. Thus Hoare logic is complete relative to the existence of an oracle for proving any true statement of the form $P \Rightarrow \texttt{wlp(}C\texttt{,}Q\texttt{)}$.

To summarise: relative completeness says that if $\texttt{wlp(}C\texttt{,}Q\texttt{)}$ is expressible in the specification language and if there is an oracle to prove true statements of the form $P \Rightarrow \texttt{wlp(}C\texttt{,}Q\texttt{)}$, then any true Hoare triple $\{P\}\ C\ \{Q\}$ can be proved using Hoare logic.

We now run through the relative completeness argument again, but in a bit more detail, showing how assumptions P_2 and P_3 are used. Recall:

$$\mathsf{Ssem}\ (\mathtt{wlp}(C\mathtt{,}Q)) \ = \ \mathsf{Wlp}\ (\mathsf{Csem}\ C)(\mathsf{Ssem}\ Q) \tag{P$_2$}$$

$$\mathsf{Hsem}\ P\ C\ Q \ = \ \forall s.\ \mathsf{Ssem}\ P\ s \Rightarrow \mathsf{Wlp}\ (\mathsf{Csem}\ C)\ (\mathsf{Ssem}\ Q)\ s \tag{P$_3$}$$

Assume for any C and Q that $\mathtt{wlp}(C\mathtt{,}Q)$ is expressible in the specification language and also that for any P, C and Q there is an oracle to prove true statements of the form $P \Rightarrow \mathtt{wlp}(C\mathtt{,}Q)$ – i.e. if $\forall s.\ \mathsf{Ssem}\ (P \Rightarrow \mathtt{wlp}(C\mathtt{,}Q))\ s$ (statement true) then the oracle gives $\vdash P \Rightarrow \mathtt{wlp}(C\mathtt{,}Q)$ (statement proved).

The Hoare triple $\{P\}\ C\ \{Q\}$ being true means, according to our semantics, that $\mathsf{Hsem}\ P\ C\ Q$ is true. If this is true, then by P_3 assumed above:

$$\forall s.\ \mathsf{Ssem}\ P\ s \Rightarrow \mathsf{Wlp}\ (\mathsf{Csem}\ C)\ (\mathsf{Ssem}\ Q)\ s$$

and then by assumed property P_2:

$$\forall s.\ \mathsf{Ssem}\ P\ s \Rightarrow \mathsf{Ssem}\ (\mathtt{wlp}(C\mathtt{,}Q))\ s$$

Although we have not completely defined the specification language, we assume at least that it contains an infix symbol $\Rightarrow$ whose meaning is logical implication, so that from the statement above we can deduce:

$$\forall s.\ \mathsf{Ssem}\ (P \Rightarrow \mathtt{wlp}(C\mathtt{,}Q))\ s$$

i.e. the statement $P \Rightarrow \mathtt{wlp}(C\mathtt{,}Q)$ is true. Now we use the assumed oracle for formulae of this form to prove $\vdash\ P \Rightarrow \mathtt{wlp}(C\mathtt{,}Q)$ and hence by assumed property P_1 and precondition strengthening, we can prove $\vdash\ \{P\}\ C\ \{Q\}$.

To complete the outline above we must define `wlp` and Wlp and prove the properties P_1, P_2 and P_3. The axioms and rules of Hoare logic will be used to prove P_1, and it is the fact that they can prove this that is really the essence of their completeness.

4.3.2 Syntactic and semantic weakest preconditions

If $P \Rightarrow Q$ we say that P is *stronger* than Q and, dually, that Q is *weaker* than P. The *weakest precondition* of a command C with respect to a postcondition Q is the weakest predicate, denoted by $\mathtt{wp}(C\mathtt{,}Q)$, such that $\mathtt{[wp(}C\mathtt{,}Q\mathtt{)]}\ C\ \mathtt{[}Q\mathtt{]}$. Notice that this is related to total correctness. The partial correctness concept is called the *weakest liberal precondition* and is denoted by $\mathtt{wlp}(C\mathtt{,}Q)$: the statement $\mathtt{wlp}(C\mathtt{,}Q)$ is the weakest predicate such that $\{\mathtt{wlp}(C\mathtt{,}Q)\}\ C\ \{Q\}$. In this chapter we only use weakest liberal preconditions. Their key properties are P_1, i.e. $\vdash\ \{\mathtt{wlp}(C\mathtt{,}Q)\}\ C\ \{Q\}$ and for all P that $\{P\}\,C\,\{Q\} \Rightarrow (P \Rightarrow \texttt{wlp(}C\texttt{,}Q\texttt{)})$. These properties can be expressed more concisely as the single equation:

$$\{P\}\,C\,\{Q\} \;=\; (P \Rightarrow \texttt{wlp(}C\texttt{,}Q\texttt{)})$$

This equation is easily seen to be equivalent to the key properties just mentioned using the rule of precondition strengthening and the reflexivity of ⇒.

If the specification language – i.e. the language of preconditions and postconditions – is strong enough to express the weakest liberal precondition for all commands C and postconditions Q then it is said to be *expressive*. Since we haven't said what the specification language is we cannot say much about expressiveness.

We can define the semantic operator Wlp on predicates via our semantics; this is an example of a *predicate transformer* [7].

$$\mathsf{Wlp}\ c\ q \;=\; \lambda s.\ \forall s'.\ c\ s\ s' \Rightarrow q\ s'$$

Recall the definition of Hsem:

$$\mathsf{Hsem}\ P\ C\ Q \;=\; \forall s_1\ s_2.\ \mathsf{Ssem}\ P\ s_1 \wedge \mathsf{Csem}\ C\ s_1\ s_2 \Rightarrow \mathsf{Ssem}\ Q\ s_2$$

We can easily prove property P_3, namely:

$$\mathsf{Hsem}\ P\ C\ Q \;=\; \forall s.\ \mathsf{Ssem}\ P\ s \Rightarrow \mathsf{Wlp}\ (\mathsf{Csem}\ P)\ (\mathsf{Ssem}\ Q)\ s$$

P_3 follows from the definitions of Hsem and Wlp by taking p, c and q to be Ssem P, Csem C and Ssem Q, respectively, in the logical truth below.

$$(\forall s_1\ s_2.\ p\ s_1 \wedge c\ s_1\ s_2 \Rightarrow q\ s_2) \;=\; (\forall s.\ p\ s \Rightarrow (\lambda s.\ \forall s'.\ c\ s\ s' \Rightarrow q\ s')\ s)$$

To prove P_1 and P_2 we need the following equations, which follow from the definition of Wlp.

$$\begin{aligned}&\mathsf{Wlp}\ (\mathsf{Csem}(V\texttt{:=}E))\ q\\ &\quad= \lambda s.\ q(s\texttt{[}(\mathsf{Esem}\ E\ s)\texttt{/V]})\end{aligned}$$

$$\begin{aligned}&\mathsf{Wlp}\ (\mathsf{Csem}(C_1\texttt{;}C_2))\ q\\ &\quad= \lambda s.\ \mathsf{Wlp}\ (\mathsf{Csem}\ C_1)\ (\mathsf{Wlp}\ (\mathsf{Csem}\ C_2)\ q)\ s\end{aligned}$$

$$\begin{aligned}&\mathsf{Wlp}\ (\mathsf{Csem}(\texttt{IF}\ S\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2))\ q\\ &\quad= \lambda s.\ \textit{if}\ \mathsf{Ssem}\ S\ s\ \textit{then}\ \mathsf{Wlp}\ (\mathsf{Csem}\ C_1)\ q\ s\ \textit{else}\ \mathsf{Wlp}\ (\mathsf{Csem}\ C_2)\ q\ s\end{aligned}$$

$$\begin{aligned}&\mathsf{Wlp}\ (\mathsf{Csem}(\texttt{WHILE}\ S\ \texttt{DO}\ C))\ q\\ &\quad= \lambda s.\ \forall n.\ \mathsf{IterWlp}\ n\ (\mathsf{Ssem}\ S)\ (\mathsf{Csem}\ C)\ q\ s\\ &\qquad\text{where}\ \mathsf{IterWlp}\ 0\ p\ c\ q\ s \quad\;\; = \neg(p\ s) \Rightarrow q\ s\\ &\qquad\qquad\quad \mathsf{IterWlp}\ (n{+}1)\ p\ c\ q\ s = p\ s \Rightarrow \mathsf{Wlp}\ c\ (\mathsf{IterWlp}\ n\ p\ c\ q)\ s\end{aligned}$$

We prove the equation for `WHILE`-commands. Expanding the definitions of Wlp and Csem yields:

$$\begin{array}{l}(\lambda s.\ \forall s'.\ (\exists n.\ \mathsf{Iter}\ n\ (\mathsf{Ssem}\ S)\ (\mathsf{Csem}\ C)\ s\ s')\ s\ s' \Rightarrow q\ s') \\ \quad = \ \lambda s.\ \forall n.\ \mathsf{IterWlp}\ n\ (\mathsf{Ssem}\ S)\ (\mathsf{Csem}\ C)\ q\ s\end{array}$$

Thus it is sufficient to prove that:

$$\forall s.\ (\forall n\ s'.\ \mathsf{Iter}\ n\ p\ c\ s\ s' \Rightarrow q\ s') \ = \ \forall n.\ \mathsf{IterWlp}\ n\ p\ c\ q\ s$$

which follows from:

$$\forall n\ s.\ (\forall s'.\ \mathsf{Iter}\ n\ p\ c\ s\ s' \Rightarrow q\ s') \ = \ \mathsf{IterWlp}\ n\ p\ c\ q\ s$$

which is equivalent to:

$$\forall n\ s.\ \mathsf{Wlp}\ (\mathsf{Iter}\ n\ p\ c)\ q\ s \ = \ \mathsf{IterWlp}\ n\ p\ c\ q\ s$$

We prove this by induction on n. First recall the definitions:

$$\begin{array}{ll}\mathsf{Iter}\ 0\ p\ c\ s_1\ s_2 & = \neg(p\ s_1) \wedge (s_1{=}s_2) \\ \mathsf{Iter}\ (n{+}1)\ p\ c\ s_1\ s_2 & = p\ s_1 \wedge \exists s.\ c\ s_1\ s \wedge \mathsf{Iter}\ n\ p\ c\ s\ s_2 \\ \mathsf{IterWlp}\ 0\ p\ c\ q & = \lambda s.\ \neg(p\ s) \Rightarrow q\ s \\ \mathsf{IterWlp}\ (n{+}1)\ p\ c\ q & = \lambda s.\ p\ s \Rightarrow \mathsf{Wlp}\ c\ (\mathsf{IterWlp}\ n\ p\ c\ q)\ s\end{array}$$

Basis.
The $n = 0$ case is $\mathsf{Wlp}\ (\mathsf{Iter}\ 0\ p\ c)\ q\ s \ = \ \mathsf{IterWlp}\ 0\ p\ c\ q\ s$ which unfolds to $(\forall s'.\ \neg(p\ s) \wedge (s = s') \Rightarrow q\ s') = \ \neg(p\ s) \Rightarrow q\ s$ which is true.

Step.
The induction hypothesis is $\forall s.\ \mathsf{Wlp}\ (\mathsf{Iter}\ n\ p\ c)\ q\ s \ = \ \mathsf{IterWlp}\ n\ p\ c\ q\ s$. From this we must show $\mathsf{Wlp}\ (\mathsf{Iter}\ (n{+}1)\ p\ c)\ q\ s \ = \ \mathsf{IterWlp}\ (n{+}1)\ p\ c\ q\ s$. This unfolds to:

$$\begin{array}{l}\mathsf{Wlp}\ (\lambda s_1\ s_2.\ p\ s_1 \ \wedge\ \exists s.\ c\ s_1\ s \ \wedge\ \mathsf{Iter}\ n\ p\ c\ s\ s_2)\ q\ s \\ \quad = \ p\ s \ \Rightarrow\ \mathsf{Wlp}\ c\ (\mathsf{IterWlp}\ n\ p\ c\ q)\ s\end{array}$$

Unfolding Wlp turns this into:

$$\begin{array}{l}(\lambda s.\ \forall s'.\ (\lambda s_1\ s_2.\ p\ s_1 \ \wedge\ \exists s.\ c\ s_1\ s \ \wedge\ \mathsf{Iter}\ n\ p\ c\ s\ s_2)\ s\ s' \Rightarrow q\ s')\ s \\ \quad = \ p\ s \ \Rightarrow\ (\lambda s.\ \forall s'.\ c\ s\ s' \Rightarrow (\mathsf{IterWlp}\ n\ p\ c\ q)\ s')\ s\end{array}$$

which reduces to:

$$\begin{array}{l}(\forall s'.\ p\ s \ \wedge\ (\exists s''.\ c\ s\ s'' \ \wedge\ \mathsf{Iter}\ n\ p\ c\ s''\ s') \Rightarrow q\ s') \\ \quad = \ p\ s \ \Rightarrow\ \forall s''.\ c\ s\ s'' \Rightarrow \mathsf{IterWlp}\ n\ p\ c\ q\ s''\end{array}$$

Using the induction hypothesis on the RHS converts this to:

$$\begin{array}{l}(\forall s'.\ p\ s \ \wedge\ (\exists s''.\ c\ s\ s'' \ \wedge\ \mathsf{Iter}\ n\ p\ c\ s''\ s') \Rightarrow q\ s') \\ \quad = \ p\ s \ \Rightarrow\ \forall s''.\ c\ s\ s'' \Rightarrow \mathsf{Wlp}\ (\mathsf{Iter}\ n\ p\ c)\ q\ s''\end{array}$$

Unfolding Wlp:

$$\begin{aligned}&(\forall s'.\ p\ s\ \wedge\ (\exists s''.\ c\ s\ s''\ \wedge\ \mathsf{Iter}\ n\ p\ c\ s''\ s') \Rightarrow q\ s')\\ &\quad=\ p\ s\ \Rightarrow\ \forall s''.\ c\ s\ s'' \Rightarrow (\lambda s.\ \forall s'.\ (\mathsf{Iter}\ n\ p\ c)\ s\ s' \Rightarrow q\ s')\ s''\end{aligned}$$

which reduces to:

$$\begin{aligned}&(\forall s'.\ p\ s\ \wedge\ (\exists s''.\ c\ s\ s''\ \wedge\ \mathsf{Iter}\ n\ p\ c\ s''\ s') \Rightarrow q\ s')\\ &\quad=\ p\ s\ \Rightarrow\ \forall s''.\ c\ s\ s'' \Rightarrow \forall s'.\ \mathsf{Iter}\ n\ p\ c\ s''\ s' \Rightarrow q\ s'\end{aligned}$$

which is true via a bit of quantifier manipulation. Thus we have proved:

$$\mathsf{Wlp}\ (\mathsf{Csem}(\texttt{WHILE}\,S\,\texttt{DO}\,C))\ q\ =\ \lambda s.\ \forall n.\ \mathsf{IterWlp}\ n\ (\mathsf{Ssem}\ S)\ (\mathsf{Csem}\ C)\ q\ s$$

4.3.3 Syntactic preconditions and expressibility

We now discuss how to define statements $\texttt{wlp(}C\texttt{,}Q\texttt{)}$ with properties P_1 and P_2, namely:

$$\vdash\ \{\texttt{wlp(}C\texttt{,}Q\texttt{)}\}\ C\ \{Q\} \qquad (\mathrm{P}_1)$$

$$\mathsf{Ssem}\ (\texttt{wlp(}C\texttt{,}Q\texttt{)})\ =\ \mathsf{Wlp}\ (\mathsf{Csem}\ C)(\mathsf{Ssem}\ Q) \qquad (\mathrm{P}_2)$$

Note that $\texttt{wlp}$ operates on syntactic things (commands and statements), whereas Wlp operates on semantic things (mathematical functions on states representing the meaning of commands and statements).

We will define $\texttt{wlp(}C\texttt{,}Q\texttt{)}$ recursively on C and justify P_1 and P_2 by structural induction on C. The cases when C is an assignment, sequence or conditional are straightforward.

$$\boxed{\texttt{wlp((}V\texttt{:=}E\texttt{),}Q\texttt{)}\ = Q\texttt{[}E/V\texttt{]}}$$

If C is $V\texttt{:=}E$ then P_1 is just the assignment axiom and by the equation for $\mathsf{Wlp}\ (\mathsf{Csem}\ (V\texttt{:=}E))\ (\mathsf{Ssem}\ Q)$ discussed on page 61, P_2 is the equation:

$$\mathsf{Ssem}\ (Q\texttt{[}E/V\texttt{]})\ s\ =\ \mathsf{Ssem}\ Q\ (s\texttt{[}(\mathsf{Esem}\ E\ s)\texttt{/}V\texttt{]})$$

which was justified in the "Diversion on substitution" on page 53.

$$\boxed{\texttt{wlp((}C_1\texttt{;}C_2\texttt{),}Q\texttt{)}\ = \texttt{wlp(}C_1\texttt{,wlp(}C_2\texttt{,}Q\texttt{))}}$$

Assume P_1 and P_2 hold for C_1 and C_2 for arbitrary Q. Then:

$$\begin{aligned}&\vdash\ \{\texttt{wlp(}C_2\texttt{,}Q\texttt{)}\}\ C_2\ \{Q\}\\ &\vdash\ \{\texttt{wlp(}C_1\texttt{,(wlp(}C_2\texttt{,}Q\texttt{)))}\}\ C_1\ \{\texttt{(wlp(}C_2\texttt{,}Q\texttt{))}\}\end{aligned}$$

hence P$_1$ by the sequencing rule. To show P$_2$ when C is C_1`;`C_2 note that P$_2$ in this case is:

$$\begin{array}{l}\mathsf{Ssem}\ (\mathtt{wlp}(C_1\mathtt{,wlp}(C_2\mathtt{,}Q\mathtt{)))}\\ \quad = \mathsf{Wlp}\ (\lambda s_1\ s_2.\ \exists s.\ \mathsf{Csem}\ C_1\ s_1\ s\ \wedge\ \mathsf{Csem}\ C_2\ s\ s_2)(\mathsf{Ssem}\ Q)\\ \quad = \lambda s_1.\ \forall s_2.\ (\exists s.\ \mathsf{Csem}\ C_1\ s_1\ s\ \wedge\ \mathsf{Csem}\ C_2\ s\ s_2) \Rightarrow \mathsf{Ssem}\ Q\ s_2\\ \quad = \lambda s_1.\ \forall s\ s_2.\ (\mathsf{Csem}\ C_1\ s_1\ s\ \wedge\ \mathsf{Csem}\ C_2\ s\ s_2) \Rightarrow \mathsf{Ssem}\ Q\ s_2\end{array}$$

Expanding the LHS using induction twice with P$_2$ instantiated with C as C_1 and Q as `wlp(`C_2`,`Q`)` and also with C as C_2 and Q just as Q gives:

$$\begin{array}{l}\mathsf{Wlp}\ (\mathsf{Csem}\ C_1)\ (\mathsf{Wlp}\ (\mathsf{Csem}\ C_2)\ (\mathsf{Ssem}\ Q))\\ \quad = \lambda s_1.\ \forall s\ s_2.\ (\mathsf{Csem}\ C_1\ s_1\ s\ \wedge\ \mathsf{Csem}\ C_2\ s\ s_2) \Rightarrow \mathsf{Ssem}\ Q\ s_2\end{array}$$

Expanding the LHS using the definition of Wlp then gives:

$$\begin{array}{l}\lambda s_1.\ \forall s.\ \mathsf{Csem}\ C_1\ s_1\ s \Rightarrow \forall s_2.\ \mathsf{Csem}\ C_2\ s\ s_2 \Rightarrow \mathsf{Ssem}\ Q\ s_2\\ \quad = \lambda s_1.\ \forall s\ s_2.\ (\mathsf{Csem}\ C_1\ s_1\ s\ \wedge\ \mathsf{Csem}\ C_2\ s\ s_2) \Rightarrow \mathsf{Ssem}\ Q\ s_2\end{array}$$

which is true.

$$\boxed{\mathtt{wlp((IF}\ S\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2\mathtt{),}Q\mathtt{)} = (S\wedge\mathtt{wlp(}C_1\mathtt{,}Q\mathtt{)})\vee(\neg S\wedge(\mathtt{wlp(}C_2\mathtt{,}Q\mathtt{))}}$$

Note that $(S \wedge S_1) \vee (\neg S \wedge S_2)$ means *if* S *then* S_1 *else* S_2. The former is used to emphasis that all we are assuming about the specification language is the existence of Boolean operators $\neg$, $\wedge$ and $\vee$. Note that by Boolean algebra and the definition of `wlp((IF` S `THEN` C_1 `ELSE` C_2`),`Q`)`:

$$\begin{array}{l}S \wedge \mathtt{wlp((IF}\ S\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2\mathtt{),}Q\mathtt{)} = S \wedge \mathtt{wlp(}C_1\mathtt{,}Q\mathtt{)}\\ \neg S \wedge \mathtt{wlp((IF}\ S\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2\mathtt{),}Q\mathtt{)} = \neg S \wedge \mathtt{wlp(}C_2\mathtt{,}Q\mathtt{)}\end{array}$$

By induction, P$_1$ for C_1 and C_2 and precondition strengthening:

$$\begin{array}{l}\{S \wedge \mathtt{wlp(}C_1\mathtt{,}Q\mathtt{)}\}\ C_1\ \{Q\}\\ \{\neg S \wedge \mathtt{wlp(}C_2\mathtt{,}Q\mathtt{)}\}\ C_2\ \{Q\}\end{array}$$

Hence by the conditional rule, substituting with the equations above:

$$\{\mathtt{wlp((IF}\ S\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2\mathtt{),}Q\mathtt{)}\}\ \mathtt{IF}\ S\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2\ \{Q\}$$

which is P$_1$ for conditionals. Property P$_2$ is:

$$\begin{array}{l}\mathsf{Ssem}\ ((S \wedge \mathtt{wlp(}C_1\mathtt{,}Q\mathtt{)}) \vee (\neg S \wedge (\mathtt{wlp(}C_2\mathtt{,}Q\mathtt{)})))\\ \quad = \mathsf{Wlp}\ (\mathsf{Csem}\ (\mathtt{IF}\ S\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2))(\mathsf{Ssem}\ Q)\end{array}$$

Expanding the RHS of this equation:

$$
\begin{array}{l}
\mathsf{Ssem}\ ((S \land \mathtt{wlp(}C_1\mathtt{,}Q\mathtt{)}) \lor (\neg S \land (\mathtt{wlp(}C_2\mathtt{,}Q\mathtt{)})))\\
\quad = \mathsf{Wlp}\\
\qquad (\lambda s_1\ s_2.\ \mathit{if}\ \mathsf{Ssem}\ S\ s_1\ \ \mathit{then}\ \ \mathsf{Csem}\ C_1\ s_1\ s_2\ \ \mathit{else}\ \ \mathsf{Csem}\ C_2\ s_1\ s_2\ )\\
\qquad (\mathsf{Ssem}\ Q)\\
\quad = \mathsf{Wlp}\\
\qquad (\lambda s_1\ s_2.\ (\mathsf{Ssem}\ S\ s_1 \land \mathsf{Csem}\ C_1\ s_1\ s_2) \lor (\neg\mathsf{Ssem}\ S\ s_1 \land \mathsf{Csem}\ C_2\ s_1\ s_2))\\
\qquad (\mathsf{Ssem}\ Q)\\
\quad = \lambda s_1.\\
\qquad \forall s_2.\ (\mathsf{Ssem}\ S\ s_1 \land \mathsf{Csem}\ C_1\ s_1\ s_2) \lor (\neg\mathsf{Ssem}\ S\ s_1 \land \mathsf{Csem}\ C_2\ s_1\ s_2)\\
\qquad\qquad \Rightarrow \mathsf{Ssem}\ Q\ s_2
\end{array}
$$

Now we expand the LHS:

$$
\begin{array}{l}
\mathsf{Ssem}\ ((S \land \mathtt{wlp(}C_1\mathtt{,}Q\mathtt{)}) \lor (\neg S \land (\mathtt{wlp(}C_2\mathtt{,}Q\mathtt{)})))\\
\quad = \lambda s_1.\ (\mathsf{Ssem}\ S\ s_1 \land \mathsf{Ssem}\ (\mathtt{wlp(}C_1\mathtt{,}Q\mathtt{)})\ s_1)\\
\qquad\qquad \lor\\
\qquad\qquad (\neg\mathsf{Ssem}\ S\ s_1 \land \mathsf{Ssem}\ (\mathtt{wlp(}C_2\mathtt{,}Q\mathtt{)})\ s_1)\\
\quad = \lambda s_1.\ (\mathsf{Ssem}\ S\ s_1 \land \mathsf{Wlp}\ (\mathsf{Csem}\ C_1)\ (\mathsf{Ssem}\ Q)\ s_1)\\
\qquad\qquad \lor\\
\qquad\qquad (\neg\mathsf{Ssem}\ S\ s_1 \land \mathsf{Wlp}\ (\mathsf{Csem}\ C_2)\ (\mathsf{Ssem}\ Q)\ s_1)\\
\quad = \lambda s_1.\ (\mathsf{Ssem}\ S\ s_1 \land (\lambda s.\ \forall s'.\ \mathsf{Csem}\ C_1\ s\ s' \Rightarrow \mathsf{Ssem}\ Q\ s')\ s_1)\\
\qquad\qquad \lor\\
\qquad\qquad (\neg\mathsf{Ssem}\ S\ s_1 \land (\lambda s.\ \forall s'.\ \mathsf{Csem}\ C_2\ s\ s' \Rightarrow \mathsf{Ssem}\ Q\ s')\ s_1)\\
\quad = \lambda s_1.\ (\mathsf{Ssem}\ S\ s_1 \land \forall s'.\ \mathsf{Csem}\ C_1\ s_1\ s' \Rightarrow \mathsf{Ssem}\ Q\ s')\\
\qquad\qquad \lor\\
\qquad\qquad (\neg\mathsf{Ssem}\ S\ s_1 \land \forall s'.\ \mathsf{Csem}\ C_2\ s_1\ s' \Rightarrow \mathsf{Ssem}\ Q\ s')
\end{array}
$$

Combining simplified LHS and RHS equations:

$$
\begin{array}{l}
\lambda s_1.\ (\mathsf{Ssem}\ S\ s_1 \land \forall s'.\ \mathsf{Csem}\ C_1\ s_1\ s' \Rightarrow \mathsf{Ssem}\ Q\ s')\\
\qquad \lor\\
\qquad (\neg\mathsf{Ssem}\ S\ s_1 \land \forall s'.\ \mathsf{Csem}\ C_2\ s_1\ s' \Rightarrow \mathsf{Ssem}\ Q\ s')\\
\quad = \lambda s_1.\\
\qquad \forall s_2.\ (\mathsf{Ssem}\ S\ s_1 \land \mathsf{Csem}\ C_1\ s_1\ s_2) \lor (\neg\mathsf{Ssem}\ S\ s_1 \land \mathsf{Csem}\ C_2\ s_1\ s_2)\\
\qquad\qquad \Rightarrow \mathsf{Ssem}\ Q\ s_2
\end{array}
$$

which is true. Thus P_2 holds for conditionals.

We are now left with defining $\mathtt{wlp((WHILE}\,S\,\mathtt{DO}\,C\mathtt{),}Q\mathtt{)}$ so that P_1 and P_2 hold. This is trickier than the previous cases. Notice that when defining $\mathtt{wlp(}C\mathtt{,}Q\mathtt{)}$ for assignments we just needed the specification language to allow textual substitution of expressions for variables and for conditionals we just needed the specification language to allow Boolean combinations of statements. The usual specification language when relative completeness is discussed is first order arithmetic. It is possible to define $\texttt{wlp((WHILE}\ S\ \texttt{DO}\ C\texttt{),}Q\texttt{)}$ for this language, but the details are fiddly. An excellent account can be found in Glynn Winskel's textbook [24, Chapter 7]. We shall instead assume more powerful features than are necessary in order to get a straightforward representation of WHILE-loop weakest preconditions. Specifically, we assume *infinite conjunctions* are allowed. What this means is that if we have an infinite family of statements, say S_n for each natural number n, then we allow an 'infinite' formula $\bigwedge n.\ S_n$ which means S_n is true for every $n \in \mathit{Num}$, i.e. $S_0 \wedge S_1 \wedge S_2 \cdots \wedge S_n \wedge \cdots$. Infinite conjunctions enable us to mimic the semantic definition in the specification language. The semantics definition is:

$$\begin{array}{l}\mathsf{Wlp}\ (\mathsf{Csem}(\texttt{WHILE}\ S\ \texttt{DO}\ C))\ q \\ \quad = \ \lambda s.\ \forall n.\ \mathsf{IterWlp}\ n\ (\mathsf{Ssem}\ S)\ (\mathsf{Csem}\ C)\ s\end{array}$$

$$\begin{array}{lll}\text{where } \mathsf{IterWlp}\ 0\ p\ c\ q & = \lambda s.\ \neg(p\ s) \Rightarrow q\ s \\ \qquad\quad \mathsf{IterWlp}\ (n{+}1)\ p\ c\ q & = \lambda s.\ p\ s \Rightarrow \mathsf{Wlp}\ c\ (\mathsf{IterWlp}\ n\ p\ c\ q)\ s\end{array}$$

the definition below in the specification language mimics this:

$$\begin{array}{l}\texttt{wlp((WHILE}\ S\ \texttt{DO}\ C\texttt{),}Q\texttt{)} \\ \quad = \ \bigwedge n.\ \texttt{iterwlp}\ n\ S\ C\ Q\end{array}$$

$$\begin{array}{ll}\text{where } \texttt{iterwlp}\ 0\ S\ C\ Q & = (\neg S \Rightarrow Q) \\ \qquad\quad \texttt{iterwlp}\ (n{+}1)\ S\ C\ Q & = (S \Rightarrow \texttt{wlp(}C\texttt{,(iterwlp}\ n\ S\ C\ Q\texttt{)))}\end{array}$$

Thus $\texttt{wlp((WHILE}\ S\ \texttt{DO}\ C\texttt{),}Q\texttt{)} = \texttt{iterwlp}\ 0\ S\ C\ Q \wedge \texttt{iterwlp}\ 1\ S\ C\ Q \cdots$ so in terms of the discussion of infinite conjunction above, we are taking S_n to be $\texttt{iterwlp}\ n\ S\ C\ Q$. In Winskel's book it is shown how Gödel's β-function[1] can be used to build a finite first order formula expressing wlp((WHILE S DO C),Q), so infinite conjunctions are not needed. However, we use the infinite formula above since it makes verifying P_1 and P_2 straightforward.

To show P_1, i.e. $\vdash\ \{\texttt{wlp((WHILE}\ S\ \texttt{DO}\ C\texttt{),}Q\texttt{)}\}\ \texttt{WHILE}\ S\ \texttt{DO}\ C\ \{Q\}$, it is sufficient to find an invariant R (perhaps provided by an annotation) such that:

$$\begin{array}{l}\vdash\ \texttt{wlp(WHILE}\ S\ \texttt{DO}\ C\texttt{,}Q\texttt{)} \Rightarrow R \\ \vdash\ R \wedge \neg S \Rightarrow Q \\ \vdash\ \{R \wedge S\}\ C\ \{R\}\end{array}$$

P_1 will then follow by the WHILE-rule and consequence rules. In fact taking R to be $\texttt{wlp(WHILE}\ S\ \texttt{DO}\ C\texttt{,}Q\texttt{)}$ will work! The first of the three conditions

[1] http://planetmath.org/encyclopedia/GodelsBetaFunction.html

above is trivial. The second is almost trivial: $\texttt{iterwlp}\ 0\ S\ C\ Q$ is $\neg S \Rightarrow Q$, so $\vdash\ (\bigwedge n.\ \texttt{iterwlp}\ n\ S\ C\ Q) \Rightarrow (\neg S \Rightarrow Q)$, hence:

$$\vdash\ (\bigwedge n.\ \texttt{iterwlp}\ n\ S\ C\ Q) \wedge \neg S \Rightarrow Q$$

i.e.:

$$\vdash\ \texttt{wlp(WHILE}\ S\ \texttt{DO}\ C\texttt{,}Q\texttt{)} \wedge \neg S \Rightarrow Q$$

For the third property we have by induction, for arbitrary n:

$$\vdash\ \{\texttt{wlp(}C\texttt{,iterwlp}\ n\ S\ C\ Q\texttt{)}\}\ C\ \{\texttt{iterwlp}\ n\ S\ C\ Q\}$$

Hence by the definition of `iterwlp` and precondition strengthening:

$$\vdash\ \{(\texttt{iterwlp}\ (n{+}1)\ S\ C\ Q) \wedge S\}\ C\ \{\texttt{iterwlp}\ n\ S\ C\ Q\}$$

Applying the rule of specification conjunction *infinitely many times*:

$$\vdash\ \{\bigwedge n.\ (\texttt{iterwlp}\ (n{+}1)\ S\ C\ Q) \wedge S\}\ C\ \{\bigwedge n.\ \texttt{iterwlp}\ n\ S\ C\ Q\}$$

In general $\vdash\ (\bigwedge n.\ S_n) \Rightarrow (\bigwedge n.\ S_{n+1})$ for any infinite set of statements S_0, S_1, ... since the set of statements being conjoined in the consequent of the implication is a subset of the set being conjoined in the antecedent. Thus by precondition strengthening applied to the Hoare triple above:

$$\vdash\ \{\bigwedge n.\ (\texttt{iterwlp}\ n\ S\ C\ Q) \wedge S\}\ C\ \{\bigwedge n.\ \texttt{iterwlp}\ n\ S\ C\ Q\}$$

In general $\vdash\ (\bigwedge n.\ (S_n \wedge S)) \Leftrightarrow ((\bigwedge n.\ S_n) \wedge S)$, so by precondition strengthening:

$$\vdash\ \{(\bigwedge n.\ \texttt{iterwlp}\ n\ S\ C\ Q) \wedge S\}\ C\ \{\bigwedge n.\ \texttt{iterwlp}\ n\ S\ C\ Q\}$$

which by the definition of $\texttt{wlp(WHILE}\ S\ \texttt{DO}\ C\texttt{,}Q\texttt{)}$ is:

$$\vdash\ \{\texttt{wlp(WHILE}\ S\ \texttt{DO}\ C\texttt{,}Q\texttt{)} \wedge S\}\ C\ \{\texttt{wlp(WHILE}\ S\ \texttt{DO}\ C\texttt{,}Q\texttt{)}\}$$

This is the desired invariance property of R. We have thus proved P_1 when C is $\texttt{WHILE}\ S\ \texttt{DO}\ C$.

To show P_2, we must show:

$$\mathsf{Ssem}\ (\texttt{wlp(WHILE}\ S\ \texttt{DO}\ C\texttt{,}Q\texttt{)})\ =\ \mathsf{Wlp}\ (\mathsf{Csem}\ (\texttt{WHILE}\ S\ \texttt{DO}\ C))\ (\mathsf{Ssem}\ Q)$$

i.e.:

$$\begin{array}{l}
\mathsf{Ssem}\ (\bigwedge n.\ \texttt{iterwlp}\ n\ S\ C\ Q) \\
=\ \mathsf{Wlp}\ (\lambda s_1\ s_2.\ \exists n.\ \mathsf{Iter}\ n\ (\mathsf{Ssem}\ S)\ (\mathsf{Csem}\ C)\ s_1\ s_2)\ (\mathsf{Ssem}\ Q) \\
=\ \lambda s.\ \forall s'.\ (\lambda s_1\ s_2.\ \exists n.\ \mathsf{Iter}\ n\ (\mathsf{Ssem}\ S)\ (\mathsf{Csem}\ C)\ s_1\ s_2)\ s\ s' \Rightarrow (\mathsf{Ssem}\ Q)\ s' \\
=\ \lambda s.\ \forall s'.\ (\exists n.\ \mathsf{Iter}\ n\ (\mathsf{Ssem}\ S)\ (\mathsf{Csem}\ C)\ s\ s') \Rightarrow \mathsf{Ssem}\ Q\ s' \\
=\ \lambda s.\ \forall s'\ n.\ \mathsf{Iter}\ n\ (\mathsf{Ssem}\ S)\ (\mathsf{Csem}\ C)\ s\ s' \Rightarrow \mathsf{Ssem}\ Q\ s'
\end{array}$$

Now $\mathsf{Ssem}\ (\bigwedge n.\ \mathtt{iterwlp}\ n\ S\ C\ Q)\ s \Leftrightarrow \forall n.\ \mathsf{Ssem}\ (\mathtt{iterwlp}\ n\ S\ C\ Q)\ s$ so we need to show for arbitrary s that:

$$\begin{array}{l} \forall n.\ \mathsf{Ssem}\ (\mathtt{iterwlp}\ n\ S\ C\ Q)\ s \\ =\ \forall s'\ n.\ \mathsf{Iter}\ n\ (\mathsf{Ssem}\ S)\ (\mathsf{Csem}\ C)\ s\ s' \Rightarrow \mathsf{Ssem}\ Q\ s' \end{array}$$

We will show by induction of n that:

$$\begin{array}{l} \mathsf{Ssem}\ (\mathtt{iterwlp}\ n\ S\ C\ Q)\ s \\ =\ \forall s'.\ \mathsf{Iter}\ n\ (\mathsf{Ssem}\ S)\ (\mathsf{Csem}\ C)\ s\ s' \Rightarrow \mathsf{Ssem}\ Q\ s' \end{array}$$

This is sufficient as $\forall n.\ (P_1\ n = P_2\ n)$ implies $(\forall n.\ P_1\ n) = (\forall n.\ P_2\ n)$. Recall the definitions of Iter and `iterwlp`:

$$\begin{array}{ll} \mathsf{Iter}\ 0\ p\ c\ s_1\ s_2 & = \neg(p\ s_1) \wedge (s_1{=}s_2) \\ \mathsf{Iter}\ (n{+}1)\ p\ c\ s_1\ s_2 & = p\ s_1 \wedge \exists s.\ c\ s_1\ s \wedge \mathsf{Iter}\ n\ p\ c\ s\ s_2 \\ \mathtt{iterwlp}\ 0\ S\ C\ Q & =\ (\neg S \Rightarrow Q) \\ \mathtt{iterwlp}\ (n{+}1)\ S\ C\ Q & =\ (S \Rightarrow \mathtt{wlp(}C\mathtt{,}(\mathtt{iterwlp}\ n\ S\ C\ Q)\mathtt{))} \end{array}$$

The basis case $(n = 0)$ is:

$$\begin{array}{l} \mathsf{Ssem}\ (\mathtt{iterwlp}\ 0\ S\ C\ Q)\ s \\ =\ \forall s'.\ \mathsf{Iter}\ 0\ (\mathsf{Ssem}\ S)\ (\mathsf{Csem}\ C)\ s\ s' \Rightarrow \mathsf{Ssem}\ Q\ s' \end{array}$$

i.e.:

$$\begin{array}{l} (\neg(\mathsf{Ssem}\ S\ s') \Rightarrow \mathsf{Ssem}\ Q\ s') \\ =\ \forall s'.\ (\neg(\mathsf{Ssem}\ S\ s) \wedge (s = s')) \Rightarrow \mathsf{Ssem}\ Q\ s' \end{array}$$

This is clearly true. The induction step case is

$$\begin{array}{l} \mathsf{Ssem}\ (\mathtt{iterwlp}\ (n{+}1)\ S\ C\ Q)\ s \\ =\ \forall s'.\ \mathsf{Iter}\ (n{+}1)\ (\mathsf{Ssem}\ S)\ (\mathsf{Csem}\ C)\ s\ s' \Rightarrow \mathsf{Ssem}\ Q\ s' \end{array}$$

Unfolding Iter and `iterwlp` yields:

$$\begin{array}{l} \mathsf{Ssem}\ (S \Rightarrow \mathtt{wlp(}C\mathtt{,}(\mathtt{iterwlp}\ n\ S\ C\ Q)\mathtt{))}\ s \\ =\ \forall s'.\ (\mathsf{Ssem}\ S\ s \wedge \exists s''.\ \mathsf{Csem}\ C\ s\ s'' \wedge \mathsf{Iter}\ n\ (\mathsf{Ssem}\ S)\ (\mathsf{Csem}\ C)\ s''\ s') \\ \qquad \Rightarrow \mathsf{Ssem}\ Q\ s' \end{array}$$

Evaluating the LHS:

$$\begin{array}{l} (\mathsf{Ssem}\ S\ s \Rightarrow \mathsf{Ssem}\ (\mathtt{wlp(}C\mathtt{,}(\mathtt{iterwlp}\ n\ S\ C\ Q)\mathtt{))}\ s) \\ =\ \forall s'.\ (\mathsf{Ssem}\ S\ s \wedge \exists s''.\ \mathsf{Csem}\ C\ s\ s'' \wedge \mathsf{Iter}\ n\ (\mathsf{Ssem}\ S)\ (\mathsf{Csem}\ C)\ s''\ s') \\ \qquad \Rightarrow \mathsf{Ssem}\ Q\ s' \end{array}$$

Using P_2 by the structural induction hypothesis (note we are doing a mathematical induction on n inside the structural induction on C to prove P_2).

$$\begin{array}{l} (\mathsf{Ssem}\ S\ s \Rightarrow \mathsf{Wlp}\ (\mathsf{Csem}\ C)\ (\mathsf{Ssem}\ (\mathtt{iterwlp}\ n\ S\ C\ Q))\ s) \\ =\ \forall s'.\ (\mathsf{Ssem}\ S\ s \wedge \exists s''.\ \mathsf{Csem}\ C\ s\ s'' \wedge \mathsf{Iter}\ n\ (\mathsf{Ssem}\ S)\ (\mathsf{Csem}\ C)\ s''\ s') \\ \qquad \Rightarrow \mathsf{Ssem}\ Q\ s' \end{array}$$

Expanding Wlp:

$$\begin{array}{l}(\mathsf{Ssem}\ S\ s \Rightarrow (\lambda s.\ \forall s'.\ (\mathsf{Csem}\ C)\ s\ s' \Rightarrow (\mathsf{Ssem}\ (\mathtt{iterwlp}\ n\ S\ C\ Q))\ s')\ s)\\ =\ \forall s'.\ (\mathsf{Ssem}\ S\ s \wedge \exists s''.\ \mathsf{Csem}\ C\ s\ s'' \wedge \mathsf{Iter}\ n\ (\mathsf{Ssem}\ S)\ (\mathsf{Csem}\ C)\ s''\ s')\\ \qquad\quad \Rightarrow \mathsf{Ssem}\ Q\ s'\end{array}$$

Reducing the LHS:

$$\begin{array}{l}(\mathsf{Ssem}\ S\ s \Rightarrow \forall s'.\ \mathsf{Csem}\ C\ s\ s' \Rightarrow \mathsf{Ssem}\ (\mathtt{iterwlp}\ n\ S\ C\ Q)\ s')\\ =\ \forall s'.\ (\mathsf{Ssem}\ S\ s \wedge \exists s''.\ \mathsf{Csem}\ C\ s\ s'' \wedge \mathsf{Iter}\ n\ (\mathsf{Ssem}\ S)\ (\mathsf{Csem}\ C)\ s''\ s')\\ \qquad\quad \Rightarrow \mathsf{Ssem}\ Q\ s'\end{array}$$

The induction hypothesis for the induction on n we are doing is:

$$\begin{array}{l}\mathsf{Ssem}\ (\mathtt{iterwlp}\ n\ S\ C\ Q)\ s\\ =\ \forall s'.\ \mathsf{Iter}\ n\ (\mathsf{Ssem}\ S)\ (\mathsf{Csem}\ C)\ s\ s' \Rightarrow \mathsf{Ssem}\ Q\ s'\end{array}$$

From this and the preceding equation:

$$\begin{array}{l}(\mathsf{Ssem}\ S\ s\\ \quad \Rightarrow \forall s'.\ \mathsf{Csem}\ C\ s\ s' \Rightarrow (\forall s''.\ \mathsf{Iter}\ n\ (\mathsf{Ssem}\ S)\ (\mathsf{Csem}\ C)\ s'\ s'' \Rightarrow \mathsf{Ssem}\ Q\ s''))\\ =\ \forall s'.\ (\mathsf{Ssem}\ S\ s \wedge \exists s''.\ \mathsf{Csem}\ C\ s\ s'' \wedge \mathsf{Iter}\ n\ (\mathsf{Ssem}\ S)\ (\mathsf{Csem}\ C)\ s''\ s')\\ \qquad\quad \Rightarrow \mathsf{Ssem}\ Q\ s'\end{array}$$

Which simplifies to:

$$\begin{array}{l}(\mathsf{Ssem}\ S\ s\\ \quad \Rightarrow \forall s'\ s''.\ \mathsf{Csem}\ C\ s\ s' \Rightarrow \mathsf{Iter}\ n\ (\mathsf{Ssem}\ S)\ (\mathsf{Csem}\ C)\ s'\ s'' \Rightarrow \mathsf{Ssem}\ Q\ s'')\\ =\ \forall s'\ s''.\ (\mathsf{Ssem}\ S\ s \wedge \mathsf{Csem}\ C\ s\ s'' \wedge \mathsf{Iter}\ n\ (\mathsf{Ssem}\ S)\ (\mathsf{Csem}\ C)\ s''\ s')\\ \qquad\quad \Rightarrow \mathsf{Ssem}\ Q\ s'\end{array}$$

Switching s' and s'' in the RHS and pulling quantifiers to the front:

$$\begin{array}{l}(\forall s'\ s''.\ \mathsf{Ssem}\ S\ s\\ \qquad\quad \Rightarrow \mathsf{Csem}\ C\ s\ s' \Rightarrow \mathsf{Iter}\ n\ (\mathsf{Ssem}\ S)\ (\mathsf{Csem}\ C)\ s'\ s'' \Rightarrow \mathsf{Ssem}\ Q\ s'')\\ =\ \forall s'\ s''.\ (\mathsf{Ssem}\ S\ s \wedge \mathsf{Csem}\ C\ s\ s' \wedge \mathsf{Iter}\ n\ (\mathsf{Ssem}\ S)\ (\mathsf{Csem}\ C)\ s'\ s'')\\ \qquad\qquad \Rightarrow \mathsf{Ssem}\ Q\ s''\end{array}$$

which is true. Thus we have proved P_2 when C is `WHILE` S `DO` C. This was the last case so we have now proved P_1 and P_2 for all commands C.

4.4 Verification conditions via `wlp`

Weakest preconditions provide a way to understand verification conditions and to improve them. Recall property P_1: $\vdash\ \{\mathtt{wlp}(C,Q)\}\ C\ \{Q\}$. To prove $\{P\}\ C\ \{Q\}$ it is thus sufficient (by precondition strengthening) to prove: $\vdash\ P \Rightarrow \mathtt{wlp}(C,Q)$ and thus one can view $P \Rightarrow \mathtt{wlp}(C,Q)$ as a single 'super verification condition' for the goal $\{P\}\ C\ \{Q\}$ which is generated without having to annotate C! This works fine if C is loop-free, i.e. contains no

`WHILE`-commands. If C does contain `WHILE`-commands then $\mathtt{wlp(}C\mathtt{,}Q\mathtt{)}$ will be an infinite statement.[2] Proving such a statement will typically involve proving by induction what is essentially the verification condition for an invariant. There is thus no getting away from finding invariants! However, it is possible to use the idea of weakest preconditions to both explain and improve the verification condition method. To see how it explains verification conditions recall from page 41 that the verification condition generated by: $\{P\}\ C_1\mathtt{;}\ \ldots\ \mathtt{;}C_{n-1}\mathtt{;}V\mathtt{:=}E\ \{Q\}$ is: $\{P\}\ C_1\mathtt{;}\ \ldots\ \mathtt{;}C_{n-1}\ \{Q\mathtt{[}E/V\mathtt{]}\}$ which is $\{P\}\ C_1\mathtt{;}\ \ldots\ \mathtt{;}C_{n-1}\ \{\mathtt{wlp(}V\mathtt{:=}E\mathtt{,}Q\mathtt{)}\}$. We can generalise this observation to reduce the number of annotations needed in sequences by only requiring annotations before commands that are *not* loop-free (i.e. contain `WHILE`-commands) and then to modify the verification conditions for sequences:

Sequences

1. The verification conditions generated by

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

(where C_n contains a `WHILE`-command) are:

(a) the verification conditions generated by

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

(b) the verification conditions generated by

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

2. The verification conditions generated by

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

(where C_n is loop-free) are the verification conditions generated by

$$\{P\}\ C_1\mathtt{;}\ \ldots\ \mathtt{;}C_{n-1}\ \{\mathtt{wlp(}C_n\mathtt{,}Q\mathtt{)}\}$$

The justification of these improved verification conditions is essentially the same as that given for the original ones, but using P$_1$ rather than the as-

[2] It is possible to represent $\mathtt{wlp(WHILE}\ S\ \mathtt{DO}\ C\mathtt{,}Q\mathtt{)}$ by a finite statement in a first order theory of arithmetic, but the statement is not suitable for use in actual verifications [24].

signment axiom. However, using wlp ideas we can do even better and reduce the requirement for annotations to just invariants of WHILE-commands. The outline of the method is as follows:

- define awp(C,Q) which is similar to wlp(C,Q) except for WHILE-commands, which must be annotated;
- define a set of statements wvc(C,Q) giving the conditions needed to verify that user-annotated invariants of all WHILE-loops in C really are invariants.

It will follow from the definitions of awp and wvc that the conjunction of the statements in wvc(C,Q) entails $\{$awp(C,Q)$\}$ C $\{Q\}$. If we define $\bigwedge\mathcal{S}$ to be the conjunction of all the statements in $\mathcal{S}$, then this can be written as:

$$\vdash\ \bigwedge\mathsf{wvc}(C,Q) \Rightarrow \{\mathtt{awp}(C\mathtt{,}Q\mathtt{)}\}\ C\ \{Q\}.$$

Hence by Modus Ponens and precondition strengthening, to prove $\{P\}\ C\ \{Q\}$ it is sufficient to prove $\vdash\ \bigwedge\mathsf{wvc}(C,Q)$ and $\vdash\ P \Rightarrow$ awp(C,Q). If C is loop-free then it turns out that awp(C,Q) $=$ wlp(C,Q) and wvc(C,Q) $= \{\}$, so this method collapses to just proving $\vdash\ P \Rightarrow$ wlp(C,Q). The definitions of awp(C,Q) and wvc(C,Q) are recursive on C and are given below. It is assumed that all WHILE-commands are annotated: WHILE S DO $\{R\}$ C.

$$\begin{array}{ll}
\mathsf{awp}(V\ \mathtt{:=}\ E,Q) & = Q\mathtt{[}E/V\mathtt{]}\\
\mathsf{awp}(C_1\ \mathtt{;}\ C_2,Q) & = \mathsf{awp}(C_1,\ \mathsf{awp}(C_2,Q))\\
\mathsf{awp}(\mathtt{IF}\ S\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2,Q) & = (S \wedge \mathsf{awp}(C_1,Q)) \vee (\neg S \wedge \mathsf{awp}(C_2,Q))\\
\mathsf{awp}(\mathtt{WHILE}\ S\ \mathtt{DO}\ \{R\}\ C,Q) & = R
\end{array}$$

$$\begin{array}{ll}
\mathsf{wvc}(V\ \mathtt{:=}\ E,Q) & = \{\}\\
\mathsf{wvc}(C_1\ \mathtt{;}\ C_2,Q) & = \mathsf{wvc}(C_1,\mathsf{awp}(C_2,Q)) \cup \mathsf{wvc}(C_2,Q)\\
\mathsf{wvc}(\mathtt{IF}\ S\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2,Q) & = \mathsf{wvc}(C_1,Q) \cup \mathsf{wvc}(C_2,Q)\\
\mathsf{wvc}(\mathtt{WHILE}\ S\ \mathtt{DO}\ \{R\}\ C,Q) & = \{R \wedge \neg S \Rightarrow Q,\ R \wedge S \Rightarrow \mathsf{awp}(C,R)\}\\
 & \quad \cup\, \mathsf{wvc}(C,R)
\end{array}$$

Theorem $\bigwedge\mathsf{wvc}(C,Q) \Rightarrow \{\mathsf{awp}(C,Q)\}\ C\ \{Q\}$.

Proof outline

Induction on C.
$C = V\mathtt{:=}E$.

$\bigwedge\mathsf{wvc}(V\mathtt{:=}E,Q) \Rightarrow \{\mathsf{awp}(C,Q)\}\ C\ \{Q\}$ is $\mathsf{T} \Rightarrow \{Q\mathtt{[}E/V\mathtt{]}\}\ V\ \mathtt{:=}\ E\ \{Q\}$
$C = C_1\mathtt{;}C_2$.

$\bigwedge\mathsf{wvc}(C_1\mathtt{;}C_2,Q) \Rightarrow \{\mathsf{awp}(C_1\mathtt{;}C_2,Q)\}\ C_1\mathtt{;}C_2\ \{Q\}$ is
$\bigwedge(\mathsf{wvc}(C_1,\mathsf{awp}(C_2,Q)) \cup \mathsf{wvc}(C_2,Q)) \Rightarrow \{\mathsf{awp}(C_1,\ \mathsf{awp}(C_2,Q))\}\ C_1\mathtt{;}C_2\ \{Q\}$.
By induction $\bigwedge\mathsf{wvc}(C_2,Q) \Rightarrow \{\mathsf{awp}(C_2,Q)\}\ C_2\ \{Q\}$
and $\bigwedge\mathsf{wvc}(C_1,\mathsf{awp}(C_2,Q)) \Rightarrow \{\mathsf{awp}(C_1,\mathsf{awp}(C_2,Q))\}\ C_1\ \{\mathsf{awp}(C_2,Q)\}$,
hence result by the Sequencing Rule.
$C = \mathtt{IF}\ S\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2$.

$\bigwedge\mathsf{wvc}(\mathtt{IF}\ S\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2, Q)$
$\Rightarrow \{\mathsf{awp}(\mathtt{IF}\ S\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2, Q)\}\ \mathtt{IF}\ S\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2\ \{Q\}$
is $\bigwedge(\mathsf{wvc}(C_1,Q) \cup \mathsf{wvc}(C_2,Q))$
$\Rightarrow \{(S\ \wedge \mathsf{awp}(C_1,Q)) \vee (\neg S \wedge \mathsf{awp}(C_2,Q)\}\ \mathtt{IF}\ S\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2\ \{Q\}$.
By induction $\bigwedge\mathsf{wvc}(C_1,Q) \Rightarrow \{\mathsf{awp}(C_1,Q)\}\ C_1\ \{Q\}$
and $\bigwedge\mathsf{wvc}(C_2,Q) \Rightarrow \{\mathsf{awp}(C_2,Q)\}\ C_2\ \{Q\}$. Strengthening preconditions gives $\bigwedge\mathsf{wvc}(C_1,Q) \Rightarrow \{\mathsf{awp}(C_1,Q) \wedge S\}\ C_1\ \{Q\}$
and $\bigwedge\mathsf{wvc}(C_2,Q) \Rightarrow \{\mathsf{awp}(C_2,Q) \wedge \neg S\}\ C_2\ \{Q\}$, hence
$\bigwedge\mathsf{wvc}(C_1,Q) \Rightarrow \{((S\ \wedge \mathsf{awp}(C_1,Q)) \vee (\neg S \wedge \mathsf{awp}(C_2,Q))) \wedge S\}\ C_1\ \{Q\}$
and $\bigwedge\mathsf{wvc}(C_2,Q) \Rightarrow \{((S\ \wedge \mathsf{awp}(C_1,Q)) \vee (\neg S \wedge \mathsf{awp}(C_2,Q))) \wedge \neg S\}\ C_2\ \{Q\}$,
hence result by the Conditional Rule.
$C = \mathtt{WHILE}\ S\ \mathtt{DO}\ C$.

$\bigwedge\mathsf{wvc}(\mathtt{WHILE}\ S\ \mathtt{DO}\ \{R\}\ C, Q) \Rightarrow \{\mathsf{awp}(\mathtt{WHILE}\ S\ \mathtt{DO}\ \{R\}\ C, Q)\}\ \mathtt{WHILE}\ S\ \mathtt{DO}\ \{R\}\ C\ \{Q\}$
is $\bigwedge(\{R \wedge \neg S \Rightarrow Q,\ R \wedge S \Rightarrow \mathsf{awp}(C,R)\} \cup \mathsf{wvc}(C,R)) \Rightarrow \{R\}\ \mathtt{WHILE}\ S\ \mathtt{DO}\ \{R\}\ C\ \{Q\}$.
By induction $\bigwedge\mathsf{wvc}(C,R) \Rightarrow \{\mathsf{awp}(C,R)\}\ C\ \{R\}$, hence result by $\mathtt{WHILE}$-Rule.

Q.E.D.

Example

$$\begin{array}{l}
\mathsf{awp}(\mathtt{R:=R-Y;Q:=Q+1},\ \mathtt{X} = \mathtt{R} + \mathtt{Y} \times \mathtt{Q}) \\
\quad = \mathsf{wlp}(\mathtt{R:=R-Y;Q:=Q+1},\ \mathtt{X} = \mathtt{R} + \mathtt{Y} \times \mathtt{Q}) \\
\quad = \mathtt{X} = \mathtt{R-Y} + \mathtt{Y} \times \mathtt{Q+1}
\end{array}$$

$$\begin{array}{l}
\mathsf{awp}(\mathtt{WHILE}\ \mathtt{Y} \leq \mathtt{R}\ \mathtt{DO}\ \{\mathtt{X} = \mathtt{R} + \mathtt{Y} \times \mathtt{Q}\}\ \mathtt{R:=R-Y;Q:=Q+1},\ \mathtt{X} = \mathtt{R}{+}\mathtt{Y}{\times}\mathtt{Q} \wedge \mathtt{R{<}Y}) \\
\quad = \mathtt{X} = \mathtt{R} + \mathtt{Y} \times \mathtt{Q}
\end{array}$$

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

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

$$\mathsf{wvc}(\mathtt{R:=R-Y;Q:=Q+1},\ \mathtt{X}) = \{\}$$

$$\begin{array}{l}
\mathsf{wvc}(\mathtt{WHILE}\ \mathtt{Y} \leq \mathtt{R}\ \mathtt{DO}\ \{\mathtt{X} = \mathtt{R} + \mathtt{Y} \times \mathtt{Q}\}\ \mathtt{R:=R-Y;Q:=Q+1},\ \mathtt{X} = \mathtt{R}{+}\mathtt{Y}{\times}\mathtt{Q} \wedge \mathtt{R{<}Y}) \\
\quad = \ \{\mathtt{X} = \mathtt{R} + \mathtt{Y} \times \mathtt{Q} \wedge \neg(\mathtt{Y} \leq \mathtt{R}) \Rightarrow \mathtt{X} = \mathtt{R}{+}\mathtt{Y}{\times}\mathtt{Q} \wedge \mathtt{R{<}Y}, \\
\qquad\ \ \mathtt{X} = \mathtt{R} + \mathtt{Y} \times \mathtt{Q} \wedge \mathtt{Y} \leq \mathtt{R} \Rightarrow \mathtt{X} = \mathtt{R-Y} + \mathtt{Y} \times \mathtt{Q+1}\} \cup \{\}
\end{array}$$

$$\begin{array}{l}
\mathsf{wvc}(\mathtt{Q:=0;WHILE}\ \mathtt{Y} \leq \mathtt{R}\ \mathtt{DO}\ \{\mathtt{X} = \mathtt{R} + \mathtt{Y} \times \mathtt{Q}\}\ \mathtt{R:=R-Y;}\ \ \mathtt{Q:=Q+1},\ \mathtt{X} = \mathtt{R}{+}\mathtt{Y}{\times}\mathtt{Q} \wedge \mathtt{R{<}Y}) \\
\quad = \{\} \cup \{\mathtt{X} = \mathtt{R} + \mathtt{Y} \times \mathtt{Q} \wedge \neg(\mathtt{Y} \leq \mathtt{R}) \Rightarrow \mathtt{X} = \mathtt{R}{+}\mathtt{Y}{\times}\mathtt{Q} \wedge \mathtt{R{<}Y}, \\
\qquad\qquad\ \mathtt{X} = \mathtt{R} + \mathtt{Y} \times \mathtt{Q} \wedge \mathtt{Y} \leq \mathtt{R} \Rightarrow \mathtt{X} = \mathtt{R-Y} + \mathtt{Y} \times \mathtt{Q+1}\}
\end{array}$$

$$\begin{array}{l}
\mathsf{wvc}(\mathtt{R=X;Q:=0;WHILE}\ \mathtt{Y} \leq \mathtt{R}\ \mathtt{DO}\ \{\mathtt{X} = \mathtt{R} + \mathtt{Y} \times \mathtt{Q}\}\ \mathtt{R:=R-Y;}\ \ \mathtt{Q:=Q+1},\ \mathtt{X} = \mathtt{R}{+}\mathtt{Y}{\times}\mathtt{Q} \wedge \mathtt{R{<}Y}) \\
\quad = \{\} \cup \{\mathtt{X} = \mathtt{R} + \mathtt{Y} \times \mathtt{Q} \wedge \neg(\mathtt{Y} \leq \mathtt{R}) \Rightarrow \mathtt{X} = \mathtt{R}{+}\mathtt{Y}{\times}\mathtt{Q} \wedge \mathtt{R{<}Y}, \\
\qquad\qquad\ \mathtt{X} = \mathtt{R} + \mathtt{Y} \times \mathtt{Q} \wedge \mathtt{Y} \leq \mathtt{R} \Rightarrow \mathtt{X} = \mathtt{R-Y} + \mathtt{Y} \times \mathtt{Q+1}\}
\end{array}$$

$\mathtt{X} = \mathtt{X} + \mathtt{Y} \times \mathtt{0}$ is T so by the theorem proved above:

$$\begin{array}{ll}
\vdash & (\mathtt{X} = \mathtt{R} + \mathtt{Y} \times \mathtt{Q} \wedge \neg(\mathtt{Y} \leq \mathtt{R}) \Rightarrow \mathtt{X} = \mathtt{R}{+}\mathtt{Y}{\times}\mathtt{Q} \wedge \mathtt{R{<}Y} \\
 & \ \wedge \\
 & \ \mathtt{X} = \mathtt{R} + \mathtt{Y} \times \mathtt{Q} \wedge \mathtt{Y} \leq \mathtt{R} \Rightarrow \mathtt{X} = \mathtt{R-Y} + \mathtt{Y} \times \mathtt{Q+1}) \\
 & \Rightarrow \\
 & \{\mathsf{T}\}\ \mathtt{R=X;Q:=0;WHILE}\ \mathtt{Y} \leq \mathtt{R}\ \mathtt{DO}\ \{\mathtt{X} = \mathtt{R} + \mathtt{Y} \times \mathtt{Q}\}\ \{\mathtt{X} = \mathtt{R}{+}\mathtt{Y}{\times}\mathtt{Q} \wedge \mathtt{R{<}Y}\}
\end{array}$$

The calculation of awp(C,Q) and wvc(C,Q) is not that different from classical verification condition generation, but has the advantage of requiring fewer annotations.

4.4.1 Strongest postconditions

Weakest preconditions are calculated 'backwards' starting from a postcondition. There is a dual theory of *strongest postconditions* that are calculated ‘forwards’ starting from a precondition. The strongest postcondition $\texttt{sp(}C\texttt{,}P\texttt{)}$ has the property that $\vdash\ \{P\}\,C\,\{\texttt{sp(}C\texttt{,}P\texttt{)}\}$ and is strongest in the sense that for any Q: if $\vdash\ \{P\}\,C\,\{Q\}$ then $\vdash\ \texttt{sp(}C\texttt{,}P\texttt{)} \Rightarrow Q$. Intuitively $\texttt{sp(}C\texttt{,}P\texttt{)}$ is a symbolic representation of the state after executing C in an initial state described by P. For assignments:

$$\texttt{sp(}V\texttt{:=}E\texttt{,}P\texttt{)} = \exists v.\ (V = E\texttt{[}v/V\texttt{]})\ \wedge\ P\texttt{[}v/V\texttt{]}\}$$

The existentially quantified variable v is the value of V in the state before executing the assignment (the initial state). The strongest postcondition expresses that after the assignment, the value of V is the value of E evaluated in the initial state (hence $E\texttt{[}v/V\texttt{]}$) and the precondition evaluated in the initial state (hence $P\texttt{[}v/V\texttt{]}$) continues to hold. Thus if the initial state is represented symbolically by the statement $(V = v) \wedge P$ then the state after executing $V\texttt{:=}E$ is represented symbolically by $(V = E\texttt{[}v/V\texttt{]})\ \wedge\ P\texttt{[}v/V\texttt{]}$.

For loop-free commands C, the calculation of $\texttt{sp(}C\texttt{,}P\texttt{)}$ amounts to the ‘symbolic execution’ of C starting from a symbolic state P. An advantages of symbolic execution is that it can allow the representation of the symbolic-state-so-far to be simplified ‘on-the-fly’, which may prune the statements generated (e.g. if the truthvalue of a conditional test can be determined then only one branch of the conditional need be symbolically executed). In the extreme case when P is so constraining that it is only satisfied by a single state, s say, then calculating $\texttt{sp(}C\texttt{,}P\texttt{)}$ collapses to just running C in s - the truthvalue of each test is determined so there is no need to consider both branches of conditionals [12]. Backwards pruning, though possible, is less natural when calculating weakest preconditions.

Several modern automatic verification methods are based on computing strongest postconditions for loop free code by symbolic execution. It is also possible to generate strongest postcondition verification conditions for `WHILE`-commands in a manner similar, but dual, to that described above using weakest preconditions. However, this is not the standard approach, though it may have future potential, especially if combined with backward methods.

4.4.2 Syntactic versus semantic proof methods

Originally Hoare logic was a proof theory for program verification that provided a method to prove programs correct by formal deduction. In practice, only simple programs could be proved by hand, and soon automated methods based on verification conditions emerged. The first idea was to convert the problem of proving $\{P\}\ C\ \{Q\}$ into a purely mathematical/logical problem of proving statements in first order logic (i.e. verification conditions) as in the early days theorem provers mainly supported first order logic. However, now we have theorem proving technology for more expressive logics (e.g. higher order logic) that are powerful enough to represent directly the semantics of Hoare triples. Thus we now have two approaches to proving $\{P\}\ C\ \{Q\}$:

(i) Syntactic: first generate VCs and then prove them;

(ii) Semantic: directly prove $\mathsf{Hsem}\ (\mathsf{Ssem}\ P)\ (\mathsf{Csem}\ C)\ (\mathsf{Ssem}\ Q)$.

Both of these approaches are used. The VC method is perhaps more common for shallow analysis of large code bases and the semantic method for full proof of correctness, though this is an oversimplification.

Chapter 5

Total Correctness

The axioms and rules of 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 2.1.8), 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 Hoare logic can be used.

5.1 Non-looping commands

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

Assignment axiom for total correctness

$$\vdash\ [P[E/V]]\ V := E\ [P]$$

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

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

Conditional rule for total correctness

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

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

$$\frac{\vdash\ \{P\}\ C\ \{Q\}}{\vdash\ [P]\ C\ [Q]} \qquad C \text{ contains no } \texttt{WHILE}\text{-commands}$$

5.2 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 := $\mathit{fact}(-1)$, where $\mathit{fact}(n)$ is defined recursively by:

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

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:

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

i.e. the assignment `X := 1/0` always halts in a state in which the condition $\mathtt{X} = 1/0$ 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 is 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.

Another approach to errors is to extend the semantics of commands to allow 'faults' to be results as well as states. This approach is used in Chapter 7 to handle memory errors, but a similar idea could also handle other expression evaluation errors (though at the expense of a more complex semantics).

5.3 `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 $\texttt{WHILE}\ S\ \texttt{DO}\ C$ 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:

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

Take

$$\begin{aligned} P &= \texttt{Y} > \texttt{0} \\ S &= \texttt{Y} \leq \texttt{R} \\ E &= \texttt{R} \\ C &= \texttt{(R:=R-Y Q:=Q+1)} \end{aligned}$$

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

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

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

and then use postcondition weakening to weaken the postcondition in the conclusion of the `WHILE`-rule to `T`. Statement (i) above is proved by showing:

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

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

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

i.e.

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

which follows from the laws of arithmetic.

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

5.4 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.

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

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

5.5 Verification conditions for termination

The idea of verification conditions is easily extended to deal with total correctness. We just consider the simple approach of Chapter 3 here, but the improved method based on weakest preconditions described in Section 4.4 is easily adapted to deal with termination.

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

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

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.

5.6 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 single verification condition for: `[X=0] X:=X+1 [X=1]` is: `X=0` $\Rightarrow$ `(X+1)=1`. This is the same as for partial correctness.

Conditionals

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 $[P\ \wedge\ \neg S]\ C_2\ [Q]$

If $C_1;\ldots;C_n$ is properly annotated, then (see page 39) 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; ... ;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

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

The only verification condition generated by this is

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

which, after doing the substitution, simplifies to

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

which is obviously true.

A correctly annotated specification of a `WHILE`-command has the form $[P]$ `WHILE` S `DO` $\{R\}[E]$ C $[Q]$

The verification conditions are:

`WHILE`-commands

The verification conditions generated from

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

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, Q.

Example: The verification conditions for

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

are:

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

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

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

together with the verification condition for

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

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)!

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

Chapter 6

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.

6.1 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 [21] and Carroll Morgan [20]), 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 4.3.3). 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 con-

[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

vey 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 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.

6.2 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 $\mathcal{C} \supseteq \mathcal{C}'$, $\mathcal{C}_1 \supseteq \mathcal{C}'_1$, $\ldots$, $\mathcal{C}_n \supseteq \mathcal{C}'_n$ then:

$$\begin{array}{ll}
\mathcal{C}_1\,;\ \cdots\ ;\mathcal{C}_n & \supseteq \mathcal{C}'_1\,;\ \cdots\ ;\mathcal{C}'_n \\[1ex]
\mathtt{BEGIN\ VAR}\ V_1\,;\ \cdots\ \mathtt{VAR}\ V_n\,;\ \mathcal{C}\ \mathtt{END} & \supseteq \mathtt{BEGIN\ VAR}\ V_1\,;\ \cdots\ \mathtt{VAR}\ V_n\,;\ \mathcal{C}'\ \mathtt{END} \\[1ex]
\mathtt{IF}\ S\ \mathtt{THEN}\ \mathcal{C} & \supseteq \mathtt{IF}\ S\ \mathtt{THEN}\ \mathcal{C}' \\[1ex]
\mathtt{IF}\ S\ \mathtt{THEN}\ \mathcal{C}_1\ \mathtt{ELSE}\ \mathcal{C}_2 & \supseteq \mathtt{IF}\ S\ \mathtt{THEN}\ \mathcal{C}'_1\ \mathtt{ELSE}\ \mathcal{C}'_2 \\[1ex]
\mathtt{WHILE}\ S\ \mathtt{DO}\ \mathcal{C} & \supseteq \mathtt{WHILE}\ S\ \mathtt{DO}\ \mathcal{C}'
\end{array}$$

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

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

Derivation

$$\begin{array}{lll}
\multicolumn{3}{l}{C \in \{\mathtt{SKIP}\}} \\
\quad\Leftrightarrow & C\ =\ \mathtt{SKIP} & \\
\quad\Rightarrow & \vdash\ [P]\ C\ [P] & \text{(Skip Axiom)} \\
\quad\Leftrightarrow & C \in [P,\ P] & \text{(Definition of } [P,\ P]\text{)}
\end{array}$$

The Assignment Law

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

Derivation

$$\begin{array}{lll}
\multicolumn{3}{l}{C \in \{V\ \mathtt{:=}\ E\}} \\
\quad\Leftrightarrow & C\ =\ V\ \mathtt{:=}\ E & \\
\quad\Rightarrow & \vdash\ [P\,\mathtt{[}E/V\mathtt{]}]\ C\ [P] & \text{(Assignment Axiom)} \\
\quad\Leftrightarrow & C \in [P\,\mathtt{[}E/V\mathtt{]},\ P] & \text{(Definition of } [P\,\mathtt{[}E/V\mathtt{]},\ P]\text{)}
\end{array}$$

Derived Assignment Law

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

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

Derivation

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

Precondition Weakening

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

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

Derivation

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

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

$$\begin{array}{lll}
\multicolumn{2}{l}{C \in [P,\ R]\ ;\ [R,\ Q]} & \\
\Leftrightarrow & C \in \{\ C_1\ ;\ C_2\ \mid\ C_1 \in [P,\ R]\ \&\ C_2 \in [R,\ Q]\} & \text{(Definition of } \mathcal{C}_1\ ;\ \mathcal{C}_2\text{)} \\
\Leftrightarrow & C \in \{\ C_1\ ;\ C_2\ \mid\ \ \vdash\ [P]\ C_1\ [R]\ \&\ \ \vdash\ [R]\ C_2\ [Q]\} & \text{(Definition of } [P,\ R] \text{ and } [R,\ Q]\text{)} \\
\Rightarrow & C \in \{\ C_1\ ;\ C_2\ \mid\ \ \vdash\ [P]\ C_1\ ;\ C_2\ [Q]\} & \text{(Sequencing Rule)} \\
\Rightarrow & \vdash\ [P]\ C\ [Q] & \\
\Leftrightarrow & C \in [P,\ Q] & \text{(Definition of } [P,\ Q]\text{)}
\end{array}$$

The Block Law

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

where V does not occur in P or Q

Derivation

$$\begin{array}{lll}
\multicolumn{2}{l}{C \in \mathtt{BEGIN\ VAR}\ V\ ;\ [P,\ Q]\ \mathtt{END}} & \\
\Leftrightarrow & C \in \{\mathtt{BEGIN\ VAR}\ V\ ;\ C'\ \mathtt{END}\ \mid & \\
 & \qquad C' \in [P,\ Q]\} & \text{(Definition of } \mathtt{BEGIN\ VAR}\ V\ ;\ \mathcal{C}\ \mathtt{END}\text{)} \\
\Leftrightarrow & C \in \{\mathtt{BEGIN\ VAR}\ V\ ;\ C'\ \mathtt{END}\ \mid & \\
 & \qquad \vdash\ [P]\ C'\ [Q]\} & \text{(Definition of } [P,\ Q]\text{)} \\
\Rightarrow & C \in \{\mathtt{BEGIN\ VAR}\ V\ ;\ C'\ \mathtt{END}\ \mid & \\
 & \qquad \vdash\ [P]\ \mathtt{BEGIN\ VAR}\ V\ ;\ C'\ \mathtt{END}\ [Q]\} & \text{(Block Rule \& } V \text{ not in } P \text{ or } Q\text{)} \\
\Rightarrow & \vdash\ [P]\ C\ [Q] & \\
\Leftrightarrow & C \in [P,\ Q] & \text{(Definition of } [P,\ Q]\text{)}
\end{array}$$

The One-armed Conditional Law

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

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

Derivation

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

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

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

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

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

6.3 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 := X ; $\mathcal{C}$ abbreviates $\{R$:= $X\}$; $\mathcal{C}$.

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

6.4 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:

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

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 7

Pointers and Local Reasoning

Reasoning about programs that manipulate pointers (e.g. in-place list reversal) can be done using Hoare logic, but with traditional methods it is cumbersome. In the last 10 years a new elegant approach based on 'local reasoning' has emerged and given rise to a version of Hoare logic called separation logic.

Programs are represented semantically as relations between initial and final states. Up to now states have been represented by functions from variables to values. To represent the pointer structures used to represent lists, trees etc. we need to add another component to states called the *heap*.

7.1 Pointer manipulation constructs

For the simple (non pointer manipulating) language in previous chapters the state was a function mapping variables to values. We now need to add a representation of the heap. Following Yang and O'Hearn [25], a *store* is defined to be what previously we called the state.[1] The set *Store* of stores is thus defined by:

$\mathit{Store} = Var \to \mathit{Val}$

Pointers will be represented by *locations*, which are mathematical abstractions of computer memory address and will be modelled by natural numbers. The contents of locations will be values, which are assumed to include both locations and data values, e.g. integers and nil (see later). The contents of pointers are stored in the *heap*, which is a finite function – i.e. a function with a finite domain – from natural numbers (representing pointers) to values.

$\mathit{Heap} = \mathit{Num} \rightharpoonup_{fin} \mathit{Val}$

where we use the notation $A \rightharpoonup_{fin} B$ to denote the set of finite functions from A to B. If $f : A \rightharpoonup_{fin} B$ then the domain of f is a finite subset of A

[1] In early work the store was called the *environment* [29] and it is now sometimes also called the *stack*.

denoted by $\mathsf{dom}(f)$ (or $\mathsf{dom}\ f$) and is the subset of A on which f is defined. The notation $f\,\texttt{[}b\texttt{/}a\texttt{]}$ denotes the function that is the same as f except that it maps a to b. If $a \notin \mathsf{dom}(f)$, then a is added to the domain of $f\,\texttt{[}b\texttt{/}a\texttt{]}$, thus: $\mathsf{dom}(f\,\texttt{[}b\texttt{/}a\texttt{]}) = \mathsf{dom}(f) \cup \{a\}$. The notation $f\texttt{-}a$ denotes the function obtained from f by deleting a from its domain, thus $\mathsf{dom}(f\texttt{-}a) = \mathsf{dom}(f)\backslash\{a\}$ (where $A\backslash B$ denotes the set of elements of A that are not in B). The notation $\{l_1 \mapsto v_1, \ldots, l_n \mapsto v_n\}$ denotes the finite function with domain $\{l_1, \ldots, l_n\}$ which maps l_i to v_i (for $1 \leq i \leq n$). A location, or pointer, is said to be in the heap h if it is a member of $\mathsf{dom}(h)$.

The new kind of state will be a pair (s, h) where $s \in \mathit{Store}$ and $h \in \mathit{Heap}$. To extend states to include heaps we redefine the set *State* of states to be:

$\mathit{State} = \mathit{Store} \times \mathit{Heap}$

We add to our language four new kinds of atomic commands that read from, write to, extend or shrink the heap. An important feature is that some of them can *fault*. For example, an attempt to read from a pointer that is not in the heap faults. The executions of these constructs takes place with respect to a given heap. The new commands are described below.

1. **Fetch assignments:** $V\texttt{:=[}E\texttt{]}$
   Evaluate E to get a location and then assign its contents to the variable V. Faults if the value of E is not in the heap.

2. **Heap assignments:** $\texttt{[}E_1\texttt{]:=}E_2$
   Evaluate E_1 to get a location and then store the value resulting from evaluating E_2 as its contents. Faults if the value of E_1 is not in the heap.

3. **Allocation assignments:** $V\texttt{:=cons(}E_1, \ldots, E_n\texttt{)}$
   Choose n consecutive locations that are not in the heap, say $l, l{+}1, \ldots$, extend the heap by adding these to its domain, assign l to the variable V and store the values of expressions $E_1, E_2, \ldots$ as the contents of $l, l{+}1, \ldots$. This is non-deterministic because any suitable $l, l{+}1, \ldots$ not in the heap can be chosen. Such numbers exist because the heap is finite. This never faults.

4. **Pointer disposal:** $\texttt{dispose(}E\texttt{)}$
   Evaluate E to get a pointer l (a number) and then remove this from the heap (i.e. remove it from the domain of the finite function representing the heap). Faults if l is not in the heap.

Example
Here is a nonsense sequence of assignments as a concrete illustration:

```
X:=cons(0,1,2); [X]:=Y+1; [X+1]:=Z; [X+2]:=Y+Z; Y:=[Y+Z]
```

The first assignment allocates three new pointers – say l, $l+1$, $l+2$ – at consecutive locations; the first is initialised with contents `0`, the second with `1` and the third with `2` and the variable `X` is assigned to point to l. The second command changes the contents of l to be the value of `Y+1`. The third command changes the contents of $l+1$ to be the value of `Z`. The last command changes the value of `Y` in the store to the contents in the heap of the value of the expression `Y+Z`, considered as a location; this might fault if the expression `Y+Z` evaluates to a number not in the heap.

For simplicity, expressions only depend on the state not the heap. Thus expressions like `[`E_1`]+[`E_2`]` are not allowed. In our language, which is adapted from the standard reference [25], only commands depend on the heap. Expressions denote functions from stores to values.

Pointers are used to represent data-structures such as linked lists and trees. We need to introduce some specification mechanisms to deal with these, which we will do in Section 7.3.5. First, as preparation, we consider some simple examples that illustrate subtleties that we have to face. Consider the following sequence of assignments:

```
X:=cons(0); Y:=X; [Y]:=Z; W:=[X]
```

This assigns `X` and `Y` to a new pointer, then makes the contents of this pointer be the value of `Z` and then assigns `W` to the value of the pointer. Thus intuitively we would expect that the following Hoare triple holds:

$\{\mathtt{T}\}$ `X:=cons(0); Y:=X; [Y]:=Z; W:=[X]` $\{\mathtt{W} = \mathtt{Z}\}$

How can we prove this? We need additional assignment axioms to handle fetch, store and allocation assignments. But this is not all ... how can we specify that the contents of the pointer values of `X` and `Y` are equal to the value of the expression `Y`? This is a property of the heap, so we need to be able to specify postconditions whose truth depends on the heap as well as on the state. We would also like to be able to specify preconditions on the heap so as to be able to prove things like:

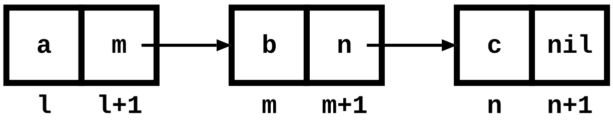
$\{$*contents of pointers* `X` *and* `Y` *are equal*$\}$ `X:=[X]; Y:=[Y]` $\{\mathtt{X} = \mathtt{Y}\}$

For example, if `X` is 1 and `Y` is 2 in the state, and if both locations 1 and 2 have contents v in the heap, then the two fetch assignments will assign v to both `X` and `Y`.

Separation logic is one of several competing methods for reasoning about pointer manipulating programs. It is a development from Hoare logic and smoothly extends the earlier material in this course. Separation logic provides various constructs for making assertions about the heap and Hoare-like axioms and rules for proving Hoare triples that use these assertions. The details are quite delicate and have taken many years to evolve, starting from work by Rod Burstall in the 1970s [27] then evolving via several only partially successful attempts until finally, reaching the current form in the work of O'Hearn, Reynolds and Yang [26] (this paper contains a short history and further references). A good introduction is John Reynolds' course notes [23], from which I have taken many ideas including the linked list reversal example in the following section.

7.2 Example: reversing a linked list

Linked lists are a simple example of a data-structure. We need to distinguish the elements of a list – the data – from the pointer structure that represents it. Each element of the list is held as the contents of a location and then the contents of the successor location is the address of the next element in the list. The end of the list is indicated by nil. The diagram below shows the list [a, b, c] stored in a linked list data-structure where a is the contents of location l, b is the contents of location m and c then contents of n. The contents of n+1 is nil, indicating the end of the list.



If `X` has value l in the store, then `X` points to a linked list holding [a, b, c].

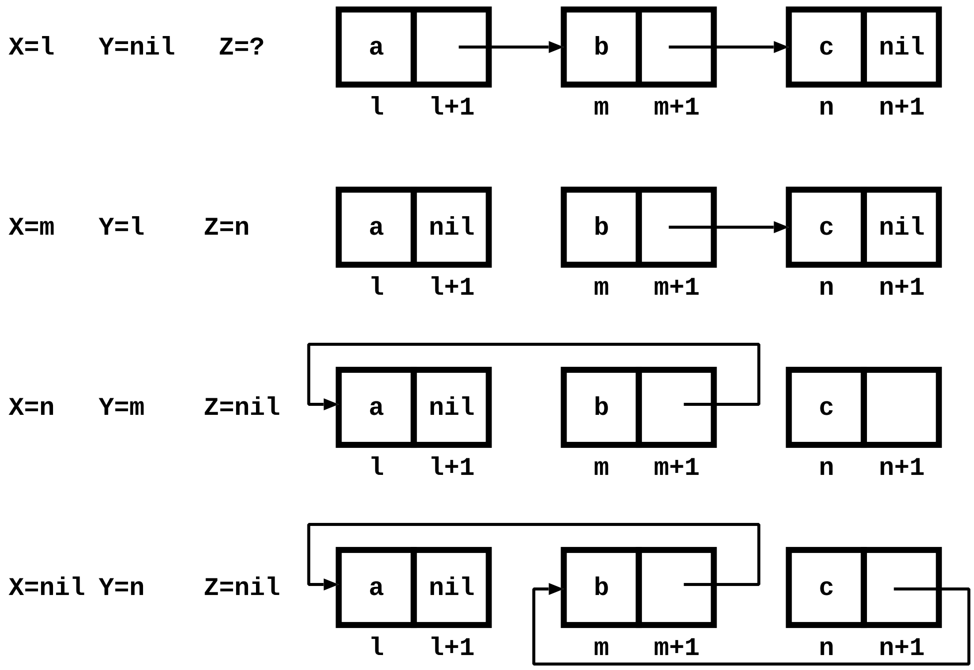
The following program reverses a linked list pointed to by `X` with the resulting reversed list being pointed to by `Y` after the loop halts.

```
Y:=nil;
WHILE ¬(X = nil) DO (Z:=[X+1]; [X+1]:=Y; Y:=X; X:=Z)
```

Below is a trace of the execution when `X` points to a linked list holding the data list [a, b, c]. A blank line precedes each loop iteration.

Store	Heap
$\mathtt{X}=l,\ \mathtt{Y}=?,\ \mathtt{Z}=?$	$l\mapsto\mathsf{a},\ l{+}1\mapsto m,\ m\mapsto\mathsf{b},\ m{+}1\mapsto n,\ n\mapsto\mathsf{c},\ n{+}1\mapsto\mathsf{nil}$
$\mathtt{X}=l,\ \mathtt{Y}=\mathsf{nil},\ \mathtt{Z}=?$	$l\mapsto\mathsf{a},\ l{+}1\mapsto m,\ m\mapsto\mathsf{b},\ m{+}1\mapsto n,\ n\mapsto\mathsf{c},\ n{+}1\mapsto\mathsf{nil}$
$\mathtt{X}=l,\ \mathtt{Y}=\mathsf{nil},\ \mathtt{Z}=m$	$l\mapsto\mathsf{a},\ l{+}1\mapsto m,\ m\mapsto\mathsf{b},\ m{+}1\mapsto n,\ n\mapsto\mathsf{c},\ n{+}1\mapsto\mathsf{nil}$
$\mathtt{X}=l,\ \mathtt{Y}=\mathsf{nil},\ \mathtt{Z}=m$	$l\mapsto\mathsf{a},\ l{+}1\mapsto\mathsf{nil},\ m\mapsto\mathsf{b},\ m{+}1\mapsto n,\ n\mapsto\mathsf{c},\ n{+}1\mapsto\mathsf{nil}$
$\mathtt{X}=l,\ \mathtt{Y}=l,\ \mathtt{Z}=m$	$l\mapsto\mathsf{a},\ l{+}1\mapsto\mathsf{nil},\ m\mapsto\mathsf{b},\ m{+}1\mapsto n,\ n\mapsto\mathsf{c},\ n{+}1\mapsto\mathsf{nil}$
$\mathtt{X}=m,\ \mathtt{Y}=l,\ \mathtt{Z}=m$	$l\mapsto\mathsf{a},\ l{+}1\mapsto\mathsf{nil},\ m\mapsto\mathsf{b},\ m{+}1\mapsto n,\ n\mapsto\mathsf{c},\ n{+}1\mapsto\mathsf{nil}$
$\mathtt{X}=m,\ \mathtt{Y}=l,\ \mathtt{Z}=n$	$l\mapsto\mathsf{a},\ l{+}1\mapsto\mathsf{nil},\ m\mapsto\mathsf{b},\ m{+}1\mapsto n,\ n\mapsto\mathsf{c},\ n{+}1\mapsto\mathsf{nil}$
$\mathtt{X}=m,\ \mathtt{Y}=l,\ \mathtt{Z}=n$	$l\mapsto\mathsf{a},\ l{+}1\mapsto\mathsf{nil},\ m\mapsto\mathsf{b},\ m{+}1\mapsto l,\ n\mapsto\mathsf{c},\ n{+}1\mapsto\mathsf{nil}$
$\mathtt{X}=m,\ \mathtt{Y}=m,\ \mathtt{Z}=n$	$l\mapsto\mathsf{a},\ l{+}1\mapsto\mathsf{nil},\ m\mapsto\mathsf{b},\ m{+}1\mapsto l,\ n\mapsto\mathsf{c},\ n{+}1\mapsto\mathsf{nil}$
$\mathtt{X}=n,\ \mathtt{Y}=m,\ \mathtt{Z}=n$	$l\mapsto\mathsf{a},\ l{+}1\mapsto\mathsf{nil},\ m\mapsto\mathsf{b},\ m{+}1\mapsto l,\ n\mapsto\mathsf{c},\ n{+}1\mapsto\mathsf{nil}$
$\mathtt{X}=n,\ \mathtt{Y}=m,\ \mathtt{Z}=\mathsf{nil}$	$l\mapsto\mathsf{a},\ l{+}1\mapsto\mathsf{nil},\ m\mapsto\mathsf{b},\ m{+}1\mapsto l,\ n\mapsto\mathsf{c},\ n{+}1\mapsto\mathsf{nil}$
$\mathtt{X}=n,\ \mathtt{Y}=m,\ \mathtt{Z}=\mathsf{nil}$	$l\mapsto\mathsf{a},\ l{+}1\mapsto\mathsf{nil},\ m\mapsto\mathsf{b},\ m{+}1\mapsto l,\ n\mapsto\mathsf{c},\ n{+}1\mapsto m$
$\mathtt{X}=n,\ \mathtt{Y}=n,\ \mathtt{Z}=\mathsf{nil}$	$l\mapsto\mathsf{a},\ l{+}1\mapsto\mathsf{nil},\ m\mapsto\mathsf{b},\ m{+}1\mapsto l,\ n\mapsto\mathsf{c},\ n{+}1\mapsto m$
$\mathtt{X}=\mathsf{nil},\ \mathtt{Y}=n,\ \mathtt{Z}=\mathsf{nil}$	$l\mapsto\mathsf{a},\ l{+}1\mapsto\mathsf{nil},\ m\mapsto\mathsf{b},\ m{+}1\mapsto l,\ n\mapsto\mathsf{c},\ n{+}1\mapsto m$

Below is a pointer diagram that shows the states at the start of each of the three iterations and the final state. The bindings of `X`, `Y` and `Z` in the store are shown to the left. The heap is to the right; addresses (locations) of the 'cons cell' boxes are shown below them.



To specify that the reversing program works we will formulate a Hoare triple that, intuitively, says:

```
{X points to a linked list holding x}
Y:=nil;
WHILE ¬(X = nil) DO (Z:=[X+1]; [X+1]:=Y; Y:=X; X:=Z)
{Y points to a linked list holding rev(x)}
```

where x is an auxiliary variable representing a list (e.g. $[\mathsf{a},\mathsf{b},\mathsf{c}]$) and $\mathsf{rev}(x)$ is the reversed list (e.g. $[\mathsf{c},\mathsf{b},\mathsf{a}]$). This is formalised using separation logic assertions, which are described in the next section.

7.3 Separation logic assertions

In Section 4.2, the semantics of a statement was represented by a predicate on states, where what we called states in that section are called stores here. We will call such statements *classical statements*. They correspond to functions of type *Store* → *Bool* and say nothing about the heap. The set of classical statements is *Sta*. For the current setting we need to redefine Ssem to map stores (rather than states) to Booleans (i.e. $\mathsf{Ssem} : \mathit{Sta} \to \mathit{Store} \to \mathit{Bool}$).

Separation logic [26] introduces a Hoare triple $\{P\}\ C\ \{Q\}$ where P and Q are predicates on the state and the state is a store-heap pair (s, h). The function SSsem maps a separation logic statement to a predicate on states. Thus if *SSta* is the set of separation logic statements (which we haven't yet described) then:

$$\mathsf{SSsem} : \mathit{SSta} \to \mathit{State} \to \mathit{Bool}$$

We call separation logic statements *separation statements*.

A classical statement S can then be regarded as a separation statement by defining:

$$\mathsf{SSsem}\ S\ (s, h)\ =\ \mathsf{Ssem}\ S\ s$$

We now describe the separation statements that do depend on the heap. In what follows, E and F are expressions, which don't depend on the heap and have semantics given by Esem, which we assume maps expressions to functions on stores (i.e. $\mathsf{Esem} : \mathit{Exp} \to \mathit{Store} \to \mathit{Val}$). P and Q will range over separation statements with semantics given by SSsem. We will give the semantics by first defining operators on the meanings of expressions and the meanings of statements and then, using these operators, define the meanings of formulae. The variables e and f will range over the meanings of expressions and p and q over the meanings of statements. Thus E and F have type *Exp* but e and f have type $\mathit{Store} \rightarrow \mathit{Val}$. Similarly P and Q have type *SSta*, but p and q have type $\mathit{State} \rightarrow \mathit{Bool}$.

In what follows we sometimes use Boolean operators that have been 'lifted' to act pointwise on properties, e.g. if p and q are properties of the state (i.e. $p : \mathit{State} \rightarrow \mathit{Bool}$ and $q : \mathit{State} \rightarrow \mathit{Bool}$) then we overload $\neg$, $\wedge$, $\vee$ and $\Rightarrow$ by defining:

$$\begin{array}{lcl}
\neg p & = & \lambda \mathit{state}.\ \neg(p\ \mathit{state}) \\
p \wedge q & = & \lambda \mathit{state}.\ p\ \mathit{state} \wedge q\ \mathit{state} \\
p \vee q & = & \lambda \mathit{state}.\ p\ \mathit{state} \vee q\ \mathit{state} \\
p \Rightarrow q & = & \lambda \mathit{state}.\ p\ \mathit{state} \Rightarrow q\ \mathit{state}
\end{array}$$

where the occurrence of $\neg$, $\wedge$, $\vee$ and $\Rightarrow$ on the left of these equations is lifted to operate on predicates and the occurrence on the right is the normal Boolean operator. The lifted operators can be used to give semantics to corresponding specification language constructs:

$$\begin{array}{lcl}
\mathsf{SSsem}\ (\neg P) & = & \neg(\mathsf{SSsem}\ P) \\
\mathsf{SSsem}\ (P \wedge Q) & = & (\mathsf{SSsem}\ P) \wedge (\mathsf{SSsem}\ Q) \\
\mathsf{SSsem}\ (P \vee Q) & = & (\mathsf{SSsem}\ P) \vee (\mathsf{SSsem}\ Q) \\
\mathsf{SSsem}\ (P \Rightarrow Q) & = & (\mathsf{SSsem}\ P) \Rightarrow (\mathsf{SSsem}\ Q)
\end{array}$$

Defining quantifiers for the specification language is slightly subtle. If P is a separation statement (normally one containing an occurrence of the variable X, though this is not required), then we can form statements $\forall X.\ P$, $\exists X.\ P$ with meaning given by:

$$\begin{array}{lcl}
\mathsf{SSsem}\ (\forall X.\ P)\ (s,h) & = & \forall v.\ \mathsf{SSsem}\ P\ (s\mathtt{[}v\mathtt{/}X\mathtt{]},h) \\
\mathsf{SSsem}\ (\exists X.\ P)\ (s,h) & = & \exists v.\ \mathsf{SSsem}\ P\ (s\mathtt{[}v\mathtt{/}X\mathtt{]},h)
\end{array}$$

An example is $\exists X.\ E \mapsto X$ defined in the next section.

7.3.1 Points-to relation: $E \mapsto F$

$E \mapsto F$ is true in state (s,h) if the domain of h is the set containing only the value of E in s and the heap maps this value to the value of F in s.

$$\boxed{\begin{array}{l}
(e \mapsto f)\ (s,h)\ =\ (\mathsf{dom}\ h = \{e\ s\})\ \wedge\ (h(e\ s) = f\ s) \\[1ex]
\mathsf{SSsem}\ (E \mapsto F)\ =\ (\mathsf{Esem}\ E) \mapsto (\mathsf{Esem}\ F)
\end{array}}$$

The first definition in the box above defines a semantic operator $\mapsto$ and the section definition uses this operator to give the semantics of formulae of the form $E \mapsto F$. Subsequent definitions will have this form.

Example
The assertion $\mathtt{X} \mapsto \mathtt{Y}{+}1$ is true for heap $\{20 \mapsto 43\}$ if in the store $\mathtt{X}$ has value 20 and $\mathtt{Y}$ has value 42.

Points-to assertions specify the contents of exactly one location in the heap. Thus (using lifted $\wedge$):

$$\begin{array}{l}
(e_1 \mapsto f_1 \wedge e_2 \mapsto f_2)(s,h) = \\
\ (\mathsf{dom}\ h = \{e_1\ s\})\ \wedge\ (h(e_1\ s) = f_1\ s) \\
\ \wedge \\
\ (\mathsf{dom}\ h = \{e_2\ s\})\ \wedge\ (h(e_2\ s) = f_2\ s)
\end{array}$$

Thus if $e_1 \mapsto f_1 \wedge e_2 \mapsto f_2$ is true in a state (s,h) then $e_1\ s = e_2\ s$ and $f_1\ s = f_2\ s$.

Abbreviation
We define $E \mapsto \_$ so that it is true of a state (s,h) when h is any heap whose domain is the singleton set $\{\mathsf{Esem}\ E\ s\}$.

$E \mapsto \_ \ = \ \exists X.\ E \mapsto X$	(where X does not occur in E)

Using the semantics of "$\exists X$" given earlier, and assuming that if X doesn't occur in E then $\mathsf{Esem}\ E\ (s[v/X]) = \mathsf{Esem}\ E\ s$, we have:

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{SSsem}\ (E \mapsto \_)\ (s,h)} \\
= & \mathsf{SSsem}\ (\exists X.\ E \mapsto X)\ (s,h) \\
= & \exists v.\ \mathsf{SSsem}\ (E \mapsto X)\ (s[v/X],h) \\
= & \exists v.\ (\mathsf{Esem}\ E \mapsto \mathsf{Esem}\ X)\ (s[v/X],h) \\
= & \exists v.\ (\mathsf{dom}\ h = \{\mathsf{Esem}\ E\ (s[v/X])\})\ \wedge \\
 & \qquad (h(\mathsf{Esem}\ E\ (s[v/X])) = \mathsf{Esem}\ X\ (s[v/X])) \\
= & \exists v.\ (\mathsf{dom}\ h = \{\mathsf{Esem}\ E\ s\})\ \wedge\ (h(\mathsf{Esem}\ E\ s) = v) \\
= & (\mathsf{dom}\ h = \{\mathsf{Esem}\ E\ s\}) \wedge \exists v.\ h(\mathsf{Esem}\ E\ s) = v \\
= & (\mathsf{dom}\ h = \{\mathsf{Esem}\ E\ s\}) \wedge \mathsf{T} \\
= & (\mathsf{dom}\ h = \{\mathsf{Esem}\ E\ s\})
\end{array}$$

which shows that $E \mapsto \_$ is true of a state (s,h) when h is any heap whose domain is $\{\mathsf{Esem}\ E\ s\}$.

The separating conjunction operator $\star$ defined below can be used to combine points-to assertions to specify heaps with bigger (i.e. non-singleton) domains.

7.3.2 Separating conjunction: $P \star Q$

Before defining the semantics of $P \star Q$ we need some preparatory definitions concerning the combination of heaps with disjoint domains.

If h_1 and h_2 are heaps then define $\mathsf{Sep}\, h_1\, h_2\, h$ to be true if and only if the domains of h_1 and h_2 are disjoint, their union is the domain of h and the contents specified by h of a location $l \in \mathsf{dom}\ h$ (i.e. $h\ l$) is the contents specified by h_1 (i.e. $h_1\ l$) if $l \in \mathsf{dom}\ h_1$ and is the contents specified by h_2 (i.e. $h_2\ l$) if $l \in \mathsf{dom}\ h_2$. This is perhaps clearer when specified formally:

$$\begin{array}{l}
\mathsf{Sep}\, h_1\, h_2\, h \ = \\
\ ((\mathsf{dom}\ h_1) \cap (\mathsf{dom}\ h_2) = \{\}) \\
\ \wedge \\
\ ((\mathsf{dom}\ h_1) \cup (\mathsf{dom}\ h_2) = (\mathsf{dom}\ h)) \\
\ \wedge \\
\ \forall l \in \mathsf{dom}\ h.\ h\ l \ = \ \mathit{if}\ l \in \mathsf{dom}\ h_1\ \ \mathit{then}\ \ h_1\ l\ \ \mathit{else}\ \ h_2\ l
\end{array}$$

The relation $\mathsf{Sep}\, h_1\, h_2\, h$ is usually written $h_1 \star h_2 = h$, where $\star$ is a partial operator that is only defined on heaps with disjoint domains.

If $(\mathsf{dom}\ h_1) \cap (\mathsf{dom}\ h_2) = \{\}$, then $h_1 \star h_2$ is defined to be the union of h_1 and h_2, i.e.:

$$\forall l \in (\mathsf{dom}\ h_1 \cup \mathsf{dom}\ h_2).\ (h_1 \star h_2)\ l \ = \ \mathit{if}\ l \in \mathsf{dom}\ h_1\ \ \mathit{then}\ \ h_1\ l\ \ \mathit{else}\ \ h_2\ l$$

Separating conjunction also uses the $\star$-symbol, but as an operator to combine separation properties: $P \star Q$ is true in state (s, h) if there exist h_1 and h_2 such that $\mathsf{Sep}\, h_1\, h_2\, h$ and P is true in state (s, h_1) and Q is true in (s, h_2). We first define a semantic version: $p \star q$ where p and q are predicates on states and then define the specification combining operator using this.

$$\boxed{\begin{array}{lcl}
(p \star q)\ (s,h) & = & \exists h_1\ h_2.\ \mathsf{Sep}\, h_1\, h_2\, h\ \wedge\ p\ (s,h_1)\ \wedge\ q\ (s,h_2) \\[1ex]
\mathsf{SSsem}\ (P \star Q) & = & (\mathsf{SSsem}\ P) \star (\mathsf{SSsem}\ Q)
\end{array}}$$

Note that the symbol $\star$ is used with three meanings: to combine heaps $(h_1 \star h_2)$, to combine semantic predicates $(p \star q)$ and to combine separation statements $(P \star Q)$.

Example

The assertion $\mathtt{X} \mapsto 0 \star \mathtt{X}{+}1 \mapsto 0$ is true of the heap $\{20 \mapsto 0, 21 \mapsto 0\}$ if $\mathtt{X}$ has value 20 in the store.

Abbreviation

The following notation defines the contents of a sequence of contiguous locations starting at the value of E to hold the values of $F_0, \ldots, F_n$.

$$E \mapsto F_0, \ldots, F_n \;=\; (E \mapsto F_0) \star \cdots \star (E{+}n \mapsto F_n)$$

Example

$\mathtt{X} \mapsto \mathtt{Y}, \mathtt{Z}$ specifies that if l is the value of $\mathtt{X}$ in the store, then heap locations l and $l{+}1$ holds the values of $\mathtt{Y}$ and $\mathtt{Z}$, respectively.

We can also define a 'semantic' version of the notation which operates on functions:

$$\begin{aligned}
&e \mapsto f_0, \ldots, f_n \;=\; (e \mapsto f_0) \star \cdots \star ((\lambda s.\ (e\ s){+}n) \mapsto f_n)\\
&\mathsf{SSsem}\ (E \mapsto F_0, \ldots, F_n) \;=\; \mathsf{Esem}\ E \mapsto (\mathsf{Esem}\ F_0), \ldots, (\mathsf{Esem}\ F_n)
\end{aligned}$$

7.3.3 Empty heap: emp

The atomic property emp is true in a state (s, h) if and only if h is the empty heap (i.e. has empty domain).

$$\begin{aligned}
&\mathsf{emp}\ (s, h) \;=\; (\mathsf{dom}\ h = \{\})\\
&\mathsf{SSsem}\ \mathsf{emp} \;=\; \mathsf{emp}
\end{aligned}$$

Example

If P is a classical property (i.e. doesn't depend on the heap) then the formula $P \wedge \mathsf{emp}$ is true iff P holds and the heap is empty.

Abbreviation

We define $E \doteq F$ to mean that E and F have equal values and the heap is empty. We also define a semantic version.

$$\begin{aligned}
(e \doteq f) &\;=\; \lambda(s, h).\ (e\ s = f\ s) \wedge (\mathsf{dom}\ h = \{\})\\
(E \doteq F) &\;=\; (E = F)\ \wedge\ \mathsf{emp}
\end{aligned}$$

From these definitions it follows that:

$$\mathsf{SSsem}\ (E \doteq F) \;=\; ((\mathsf{Esem}\ E) \doteq (\mathsf{Esem}\ F)).$$

It also follows from the semantics that:

$$\begin{aligned}
&\forall s\ h.\ \mathsf{SSsem}\ ((E \doteq F) \star P)\ (s, h) \;=\\
&\qquad (\mathsf{Esem}\ E\ s = \mathsf{Esem}\ F\ s)\ \wedge\ \mathsf{Ssem}\ P\ (s, h)
\end{aligned}$$

Using lifted $\wedge$ notation, we can write: $(e \doteq f) \star p \;=\; (e = f) \wedge p$.

7.3.4 Separating implication: $P \mathrel{-\!\!*} Q$

$P \mathrel{-\!\!*} Q$ is true in a state (s,h) if whenever P holds of a state (s,h'), where h' is disjoint from h then Q holds for the state $(s, h \star h')$ in which the heap h is extended by h'.

$$(p \mathrel{-\!\!*} q)\ (s,h) \ = \ \forall h'\ h''.\ \mathsf{Sep}\ h\ h'\ h'' \ \wedge \ p\ (s,h') \Rightarrow q\ (s,h'')$$

$$\mathsf{SSsem}\ (P \mathrel{-\!\!*} Q) \ = \ (\mathsf{SSsem}\ P) \mathrel{-\!\!*} (\mathsf{SSsem}\ Q)$$

We do not use separating implication here, but are mentioning it as it is a standard part of separation logic.

7.3.5 Formal definition of linked lists

If α is a list (e.g. $\mathtt{[}\mathsf{a},\mathsf{b},\mathsf{c}\mathtt{]}$) and e is the meaning of an expression (i.e. a function from stores to values) then $\mathsf{list}\ \alpha\ e\ (s,h)$ is defined to mean that α is represented as a linked list in the heap h starting at the location specified by $e\ s$. The definition is by structural recursion on α:

$$\mathsf{list}\ \mathtt{[]}\ e \ = \ (e \doteq \mathsf{nil})$$

$$\mathsf{list}\ ([a_0, a_1, \ldots, a_n])\ e \ = \ \exists e'.\ (e \mapsto a_0, e') \ \star \ \mathsf{list}\ [a_1, \ldots, a_n]\ e'$$

Let*List*$[X]$ be the set of lists whose elements are in X. The meaning of *List*$[X]$ is somewhat analogous to the meaning of the regular expression $X^\star$. Here is type of the function list:

$$\mathsf{list} : \mathit{List}[\mathit{Val}] \to (\mathit{Store} \to \mathit{Val}) \to \mathit{State} \to \mathit{Bool}$$

The definition of list above defines a semantic operator. We also use list to formulate separation properties.

$$\mathsf{SSsem}\ (\mathsf{list}\ \alpha\ E) \ = \ \mathsf{list}\ \alpha\ (\mathsf{Esem}\ E)$$

where the occurrence of list on the left of this definition is part of the specification language and the occurrence on the right is the semantic operator.

Recall the informal Hoare triple given earlier to specify the list reversing function.

```
{X points to a linked list holding $\alpha_0$}
Y:=nil;
WHILE ¬(X = nil) DO (Z:=[X+1]; [X+1]:=Y; Y:=X; X:=Z)
{Y points to a linked list holding rev($\alpha_0$)}
```

Using separation logic this can be formalised as:

$\{\mathsf{list}\ \alpha_0\ \texttt{X}\}$
`Y:=nil;`
`WHILE` $\neg$`(X = nil) DO (Z:=[X+1]; [X+1]:=Y; Y:=X; X:=Z)`
$\{\mathsf{list}\ (\mathsf{rev}(\alpha_0))\ \texttt{Y}\}$

and the invariant for the `WHILE`-loop turns out to be:

$$\exists\alpha\ \beta.\ \mathsf{list}\ \alpha\ \texttt{X} \star \mathsf{list}\ \beta\ \texttt{Y}\ \wedge\ (\mathsf{rev}(\alpha_0) = \mathsf{rev}(\alpha)\cdot\beta)$$

where "·" is the list concatenation operator (later we also use it for list 'cons').

7.4 Semantics and separation logic

In this section we give both semantics for the extended programming language and for Hoare logic axioms and rules for reasoning about it.

As heap operations may fault, we define the set *Result* of results of command executions to be:

$\mathit{Result} = \mathit{State} \cup \{\mathsf{fault}\}$ (where it is assumed that $\mathsf{fault} \notin \mathit{State}$)

and then the semantic function for commands, Csem, will have the more general type:

$\mathsf{Csem} : \mathit{Com} \to \mathit{State} \to \mathit{Result} \to \mathit{Bool}$

and now $\mathsf{Csem}\ C\ (s,h)\ r$ will mean that if C is started in state (s,h) then r is a possible result. As mentioned earlier, we assume that expressions do not depend on the heap, only on the store. We also assume this about classical statements. Furthermore, the evaluation of neither of these can fault, thus we redefine:

$\mathsf{Esem} : \mathit{Exp} \to \mathit{Store} \to \mathit{Val}$
$\mathsf{Ssem} : \mathit{Sta} \to \mathit{Store} \to \mathit{Bool}$

For comparison, here are the various types and semantic functions for the previous simple semantics and then for the new heap semantics.

Simple semantics (state maps variables to values)

$\mathit{State} = \mathit{Var} \to \mathit{Val}$

$\mathsf{Esem} : \mathit{Exp} \to \mathit{State} \to \mathit{Val}$
$\mathsf{Ssem} : \mathit{Sta} \to \mathit{State} \to \mathit{Bool}$
$\mathsf{Csem} : \mathit{Com} \to \mathit{State} \to \mathit{State} \to \mathit{Bool}$

Heap semantics (state is store and heap)

$\mathit{Store} = \mathit{Var} \to \mathit{Val}$ (assume $\mathit{Num} \subseteq \mathit{Val}$, $\mathsf{nil} \in \mathit{Val}$ and $\mathsf{nil} \notin \mathit{Num}$)
$\mathit{Heap} = \mathit{Num} \rightharpoonup_{\mathit{fin}} \mathit{Val}$
$\mathit{State} = \mathit{Store} \times \mathit{Heap}$
$\mathit{Result} = \mathit{State} \cup \{\mathsf{fault}\}$ (assume $\mathsf{fault} \notin \mathit{State}$)

$\mathsf{Esem} : \mathit{Exp} \to \mathit{Store} \to \mathit{Val}$
$\mathsf{Ssem} : \mathit{Sta} \to \mathit{Store} \to \mathit{Bool}$ (classical statements)
$\mathsf{SSsem} : \mathit{Sta} \to \mathit{State} \to \mathit{Bool}$ (separation statements)
$\mathsf{Csem} : \mathit{Com} \to \mathit{State} \to \mathit{Result} \to \mathit{Bool}$

The meaning of Hoare triples $\{P\}\ C\ \{Q\}$ is subtly, but very significantly, different for separation logic: it is required that for the triple to be true the execution of C in a state satisfying P *must not fault*, as well as Q holding in the final state if execution terminates. Formally, the semantics of $\{P\}\ C\ \{Q\}$ for separation logic is $\mathsf{SHsem}\ P\ C\ Q$, where:

$$\begin{array}{l} \mathsf{SHsem}\ P\ C\ Q\ = \\ \ \forall s\ h.\ \mathsf{SSsem}\ P\ (s,h) \\ \qquad\quad \Rightarrow \\ \qquad\quad \neg(\mathsf{Csem}\ C\ (s,h)\ \mathsf{fault})\ \wedge\ \forall r.\ \mathsf{Csem}\ C\ (s,h)\ r \Rightarrow \mathsf{SSsem}\ Q\ r \end{array}$$

The function SHsem has type $\mathit{Sta} \to \mathit{Com} \to \mathit{Sta} \to \mathit{Bool}$. It is useful to define a semantic function shsem so that:

$$\mathsf{SHsem}\ P\ C\ Q\ =\ \mathsf{shsem}\ (\mathsf{SSsem}\ P)\ (\mathsf{Csem}\ C)\ (\mathsf{SSsem}\ Q)$$

The definition is just:

$$\mathsf{shsem}\ p\ c\ q\ =\ \forall s\ h.\ p(s,h) \Rightarrow \neg(c\ (s,h)\ \mathsf{fault}) \wedge \forall r.\ c\ (s,h)\ r\ \Rightarrow\ q\ r$$

The type of shsem is:

$$(\mathit{State} \to \mathit{Bool}) \to (\mathit{State} \to \mathit{Result} \to \mathit{Bool}) \to (\mathit{State} \to \mathit{Bool}) \to \mathit{Bool}$$

There are two reasons for the *non-faulting semantics* of Hoare triples:

(i) to support verifying that programs do not read or write locations not specified in the precondition – i.e. memory safety;

(ii) the non-faulting semantics is needed for the soundness of the crucial *Frame Rule* for local reasoning, which is discussed later.

Non-faulting should not be confused with non-termination: the non-faulting requirement is a safety property ("nothing bad happens") not a liveness property ("something good happens"). Separation logic can straightforwardly be extended to total correctness – a liveness property – but we do not do this.

The semantics we give here is equivalent to the large-step operational semantics of Yang and O'Hearn [25, Table 2], but presented in the denotational style used in Chapter 4 for the simple language. With the semantics given here, proofs are done by structural induction for loop-free commands plus mathematical induction for `WHILE`-commands. With an operational semantics, the equivalent same proofs are done using rule-induction.

For each construct we give the semantics followed by the separation logic axiom schemes or rules of inference. It is only the axiom schemes for the atomic commands that read or modify the heap that are new. The rules for sequences, conditionals and `WHILE`-commands remain the same (the non-faulting semantics makes their soundness justification slightly more complex).

7.4.1 Purely logical rules

From the definition of SHsem it follows that the rules of consequence, i.e. precondition strengthening and postcondition weakening are sound by logic alone: their soundness doesn't depend on the semantics of commands.

Rules of consequence

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

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

Another rule that follows from the definition of SHsem (and also from that of Hsem) is the following.

Exists introduction

$$\frac{\vdash\ \{P\}\ C\ \{Q\}}{\vdash\ \{\exists x.\ P\}\ C\ \{\exists x.\ Q\}}$$

where x does not occur in C

Although valid for ordinary Hoare logic, this is not much use there. However, it is very useful in separation logic, as we shall see in Section 7.8.

7.4.2 Semantics of store assignments

Store assignments $V\texttt{:=}E$ were in the earlier language without pointers. They ignore the heap and always succeed.

$$\boxed{\mathsf{Csem}\ (V\texttt{:=}E)\ (s,h)\ r\ =\ (r = (s\texttt{[}(\mathsf{Esem}\ E\ s)\texttt{/}V\texttt{]}, h))}$$

Note that s here ranges over stores not states, thus in the above semantic equation: $s \in \mathit{Store}$, $h \in \mathit{Heap}$, $(s,h) \in \mathit{State}$ and $r \in \mathit{Result}$.

7.4.3 Store assignment axiom

First recall the classical Hoare assignment axiom scheme:

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

Although this is sound for separation logic, it is not the axiom usually given [26] – a 'small' Floyd-style forward axiom is used instead. This style of axiom is also used for all the axioms below. Perhaps the reason for this forward 'strongest postcondition' style is because it connects more directly with symbolic execution, which is a technique widely used by program analysis tools based on separation logic.

Store assignment axiom

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

where v is an auxiliary variable not occurring in E.

Note that the meaning of $\doteq$ forces any state for which the precondition is true to have an empty heap. Store assignments do not fault, so this is sound.

If V does not occur in E, then, as $(V \doteq V) = \mathsf{emp}$ and $E\texttt{[}V\texttt{/}V\texttt{]} = E$ it follows that the following is a derived axiom:

$$\vdash\ \{\mathsf{emp}\}\ V\texttt{:=}E\ \{V \doteq E\} \qquad \text{(where } V \text{ doesn't occur in } E\text{)}$$

Another derived axiom is obtained using the exists introduction rule to obtain the following from the store assignment axiom:

$$\vdash\ \{\exists v.\ V \doteq v\}\ V\texttt{:=}E\ \{\exists v.\ V \doteq E\texttt{[}v\texttt{/}V\texttt{]}\}$$

The precondition of this is emp. This follows from the definitions of $\doteq$ and lifted quantification:

$$(\exists v.\ V \doteq v)(s,h) \ = \ \exists v.\ (s\ V = v\ s) \wedge (\mathsf{dom}\ h = \{\})$$

The statement $\exists v.\ (s\ V = v\ s)$ is true – to see this take v to be $\lambda s.\ s\ V$ – hence $(\exists v.\ V \doteq v) = \mathsf{emp}$ and so the following is a derived axiom:

$$\vdash\ \{\mathsf{emp}\}\ V\mathtt{:=}E\ \{\exists v.\ V \doteq E\mathtt{[}v/V\mathtt{]}\} \qquad \text{(where } v \text{ doesn't occur in } E\text{)}$$

7.4.4 Semantics of fetch assignments

Fetch assignments change the store with the value of a location in the heap, faulting if the location is not in the heap. They do not change the heap.

$$\begin{array}{l}\mathsf{Csem}\ (V\mathtt{:=[}E\mathtt{]})\ (s,h)\ r = \\ \ (r = \textit{if}\ \mathsf{Esem}\ E\ s \in \mathsf{dom}(h)\ \textit{then}\ (s\mathtt{[}h(\mathsf{Esem}\ E\ s)\mathtt{/}\mathsf{Esem}\ E\ s\mathtt{]}, h)\ \textit{else}\ \mathsf{fault})\end{array}$$

In the above semantic equation: $s \in \textit{Store}$, $h \in \textit{Heap}$, $(s,h) \in \textit{State}$ and $r \in \textit{Result}$.

7.4.5 Fetch assignment axiom

Fetch assignment axiom

$$\vdash\ \{(V = v_1) \wedge E \mapsto v_2\}\ V\mathtt{:=[}E\mathtt{]}\ \{(V = v_2) \wedge E\mathtt{[}v_1/V\mathtt{]} \mapsto v_2\}$$

where v_1, v_2 are auxiliary variables not occurring in E.

Like the store assignment axiom above, this is best understood as describing symbolic execution. Note that the precondition requires the heap to contain a single location given by the value of E in the store and whose contents is v_2. After the fetch assignment, the variable V has the value v_2 in the store and the heap is unchanged (because the value of $E\mathtt{[}v_1/V\mathtt{]}$ in the postcondition state is the same as the value of E in the precondition state). The precondition ensures that the fetch assignment won't fault since the value of E is specified by $E \mapsto v_2$ to be in the heap.

7.4.6 Semantics of heap assignments

Heap assignments change the value of a location in the heap, faulting if the location is not in its domain. The store is unchanged.

$$\begin{array}{l}\mathsf{Csem}\ (\mathtt{[}E_1\mathtt{]:=}E_2)\ (s,h)\ r = \\ \ (r = \textit{if}\ \mathsf{Esem}\ E_1\ s \in \mathsf{dom}(h)\ \textit{then}\ (s, h\mathtt{[}\mathsf{Esem}\ E_2\ s\mathtt{/}\mathsf{Esem}\ E_1\ s\mathtt{]})\ \textit{else}\ \mathsf{fault})\end{array}$$

7.4.7 Heap assignment axiom

Heap assignment axiom

$$\vdash\ \{E \mapsto \_\}\ \texttt{[}E\texttt{]:=}F\ \{E \mapsto F\}$$

This is another forward symbolic execution style axiom. The precondition asserts that domain of the heap consists of the value of E in the store and thus the heap assignment does not fault.

7.4.8 Semantics of allocation assignments

Allocation assignments change both the store and the heap. They non-deterministically choose n contiguous locations, say l, $l{+}1, \ldots, l{+}(n{-}1)$, that are not in the heap (where n is the number of arguments of the `cons`) and then set the contents of these new locations to be the values of the arguments of the `cons`. Allocation assignments never fault.

$$\begin{array}{l}\mathsf{Csem}\ (V\texttt{:=cons(}E_1,\ldots,E_n\texttt{)})\ (s,h)\ r = \\ \ \exists l.\ l \notin \mathsf{dom}(h)\ \wedge \cdots \wedge\ l{+}(n{-}1) \notin \mathsf{dom}(h)\ \wedge \\ \quad\ (r = (s\texttt{[}l\texttt{/}V\texttt{]},\ h\texttt{[}\mathsf{Esem}\ E_1\ s\texttt{/}l\texttt{]} \cdots \texttt{[}\mathsf{Esem}\ E_n\ s\texttt{/}l{+}(n{-}1)\texttt{]}))\end{array}$$

This is non-deterministic because $\mathsf{Csem}\ (V\texttt{:=cons(}E_1,\ldots,E_n\texttt{)})\ (s,h)\ r$ is true for any result r for which the right hand side of the equation above holds. As the heap is finite, there will be infinitely many such results.

7.4.9 Allocation assignment axioms

Allocation assignment axioms

$$\vdash\ \{V \doteq v\}\ V\texttt{:=cons(}E_1,\ldots,E_n\texttt{)}\ \{V \mapsto E_1\texttt{[}v\texttt{/}V\texttt{]},\ldots,E_n\texttt{[}v\texttt{/}V\texttt{]}\}$$

where v is an auxiliary variable not equal to V.

$$\vdash\ \{\mathsf{emp}\}\ V\texttt{:=cons(}E_1,\ldots,E_n\texttt{)}\ \{V \mapsto E_1,\ldots,E_n\}$$

where V is an auxiliary variable not occurring in E_1,. . .,E_n.

These are also forward symbolic execution style axioms – but they are non-deterministic. The preconditions assert that the heap is empty. In the first axiom, the precondition also specifies that V has value v in the store. The postconditions use the abbreviation in Section 7.3.2 for specifying a contiguous chunk of memory and asserts that the domain of the heap is n contiguous locations which contain the values of $E_1,\cdots,E_n$ in the precondition store. Notice that this axiom does not determine that value of V after the assignment – so is non-deterministic – it merely requires that V points to any location not in the heap before the command is executed.

7.4.10 Semantics of pointer disposal

Pointer disposals deallocate a location by deleting it from the heap's domain, faulting if the location isn't in the domain. The store is unchanged.

$$\begin{array}{l}
\mathsf{Csem}\ (\mathtt{dispose(}E\mathtt{)})\ (s,h)\ r = \\
\ (r = \mathit{if}\ \ \mathsf{Esem}\ E\ s \in \mathsf{dom}(h)\ \mathit{then}\,(s,h\text{-}(\mathsf{Esem}\ E\ s))\ \mathit{else}\,\mathsf{fault})
\end{array}$$

7.4.11 Dispose axiom

Dispose axiom

$$\vdash\ \{E \mapsto \_\}\ \mathtt{dispose(}E\mathtt{)}\ \{\mathsf{emp}\}$$

Requires the heap to contain only one location and then deallocates it resulting in the empty heap.

7.4.12 Semantics of sequences

If neither C_1 nor C_2 faults then the semantics of $C_1;C_2$ is as before. If either C_1 or C_2 faults, then so does $C_1;C_2$.

$$\begin{array}{l}
\mathsf{Csem}\ (C_1;C_2)\ (s,h)\ r = \\
\ \mathit{if}\,(\exists s'\ h'.\ r = (s',h')) \\
\ \ \mathit{then}\ (\exists s'\ h'.\ \mathsf{Csem}\ C_1\ (s,h)\ (s',h') \wedge \mathsf{Csem}\ C_2\ (s',h')\ r) \\
\ \ \mathit{else}\ \ ((\mathsf{Csem}\ C_1\ (s,h)\ r \wedge (r = \mathsf{fault})) \\
\qquad\quad \vee \\
\qquad\quad \exists s'\ h'.\ \mathsf{Csem}\ C_1\ (s,h)\ (s',h') \wedge \mathsf{Csem}\ C_2\ (s',h')\ r \wedge (r = \mathsf{fault}))
\end{array}$$

7.4.13 The sequencing rule

The sequencing rule is unchanged for separation logic. Note that if the hypotheses are true, then there is no faulting.

The sequencing rule

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

The proof of soundness of the sequencing rule is straightforward. The argument is similar to the one given for simple Hoare logic in Section 4.2 with some additional arguments to handle faults. One proves:

$$\begin{array}{l} \forall p\ q\ r\ c_1\ c_2. \\ \quad \mathsf{shsem}\ p\ c_1\ r \wedge \mathsf{shsem}\ r\ c_2\ q \\ \quad \Rightarrow \\ \quad \mathsf{shsem}\ p\ (\lambda(s,h)\ r.\ \exists s'\ h'.\ c_1\ (s,h)\ (s',h') \wedge c_2\ (s',h')\ r)\ q \end{array}$$

where shsem is the semantic function representing the meaning of separation logic Hoare triples which was defined on page 109.

7.4.14 Semantics of conditionals

The semantics of conditionals is as before (see Section 4.1.2).

$$\begin{array}{l} \mathsf{Csem}\ (\texttt{IF}\ S\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2)\ (s,h)\ r = \\ \ \textit{if}\ \mathsf{Ssem}\ S\ s\ \ \textit{then}\ \ \mathsf{Csem}\ C_1\ (s,h)\ r\ \ \textit{else}\ \ \mathsf{Csem}\ C_2\ (s,h)\ r \end{array}$$

7.4.15 The conditional rule

The conditional rule is unchanged.

The conditional rule

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

The proof of soundness of the conditional rule is straightforward. One proves:

$$\begin{array}{l} \forall p\ q\ b\ c_1\ c_2. \\ \quad \mathsf{shsem}\ (p \wedge b)\ c_1\ q \wedge \mathsf{shsem}\ (p \wedge \neg b)\ c_2\ q \\ \quad \Rightarrow \\ \quad \mathsf{shsem}\ p\ (\lambda(s,h)\ r.\ \textit{if}\ b(s,h)\ \ \textit{then}\ \ c_1\ (s,h)\ r\ \ \textit{else}\ \ c_2\ (s,h)\ r)\ q \end{array}$$

Notice that in $(p \wedge b)$ and $(p \wedge \neg b)$ the conjunction $\wedge$ and negation $\neg$ are lifted (see page 103).

7.4.16 Semantics of WHILE-commands

The semantics of WHILE-commands is similar to the one given in Section 4.1.2 except that if a fault arises during the execution then the iteration aborts with a fault.

$$\boxed{\mathsf{Csem}\ (\mathtt{WHILE}\ S\ \mathtt{DO}\ C)\ (s,h)\ r\ =\ \exists n.\ \mathsf{Iter}\ n\ (\mathsf{Ssem}\ S)\ (\mathsf{Csem}\ C)\ (s,h)\ r}$$

The function Iter is redefined to handle faulting:

$$\mathsf{Iter}\ 0\ p\ c\ (s,h)\ r\ =\ \neg(p\ s) \wedge (r = (s,h))$$

$$\begin{array}{l}
\mathsf{Iter}\ (n{+}1)\ p\ c\ (s,h)\ r\ = \\
\ p\ s \wedge (\mathit{if}\ (\exists s'\ h'.\ r = (s'h')) \\
\qquad\quad \mathit{then}\ (\exists s'\ h'.\ c(s,h)(s'h') \wedge \mathsf{Iter}\ n\ p\ c\ (s',h')\ r) \\
\qquad\quad \mathit{else}\ ((c\ (s,h)\ r \wedge (r = \mathsf{fault})) \\
\qquad\qquad\quad \vee \\
\qquad\qquad\quad \exists s'\ h'.\ c\ (s,h)\ (s',h') \wedge \mathsf{Iter}\ n\ p\ c\ (s',h')\ r \wedge (r = \mathsf{fault})))
\end{array}$$

The type of Iter is:

$$\mathsf{Iter} : \mathit{Num}{\to}(\mathit{Store}{\to}\mathit{Bool}){\to}(\mathit{State}{\to}\mathit{Result}{\to}\mathit{Bool}){\to}\mathit{State}{\to}\mathit{Result}{\to}\mathit{Bool}$$

7.4.17 The WHILE-rule

The WHILE-rule is unchanged.

The WHILE-rule

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

The semantics of WHILE commands is defined in terms of the function Iter. The following two lemmas about Iter are straightforward to prove by induction on n.

$$\mathsf{shsem}\ (p \wedge b)\ c\ p \Rightarrow \forall n\ s\ h.\ p(s,h) \Rightarrow \neg(\mathsf{Iter}\ n\ b\ c\ (s,h)\ \mathsf{fault})$$

$$\begin{array}{l}
\mathsf{shsem}\ (p \wedge b)\ c\ p \\
\ \Rightarrow \\
\ \forall n\ s\ h\ s'\ h'.\ p(s,h) \wedge \mathsf{Iter}\ n\ b\ c\ (s,h)\ (s',h') \Rightarrow p(s',h') \wedge \neg(b(s',h'))
\end{array}$$

Notice that in $(p \wedge b)$ the conjunction $\wedge$ is lifted. The soundness of the WHILE rule follows easily from these lemmas.

7.5 The frame rule

The frame rule is the key rule of separation logic. The motivation given here is based on the account in Reynolds' notes [23]. The purpose of the frame rule is to enable *local reasoning* about just those locations that a command reads and writes to be extended to uninvolved locations, which are unchanged. How to handle this gracefully is the so called *frame problem* that was identified 50 years ago as a problem in using logic to model actions in artificial intelligence.[2]

The following rule, which Reynolds calls the rule of constancy, holds in the simple language without a heap (the proof is by structural induction on C). A variable V is said to be *modified by C* is it occurs on the left of `:=` in a store, fetch or allocation assignment in C (variables on the left of heap assignments are not modified).

The rule of constancy

$$\frac{\vdash\ \{P\}\,C\,\{Q\}}{\vdash\ \{P \wedge R\}\,C\,\{Q \wedge R\}}$$

where no variable modified by C occurs free in R.

this is not valid for heap assignments because although by the heap assignment axiom:

$\vdash\ \{\mathtt{X} \mapsto \_\}\ \mathtt{[X]:=}0\ \{\mathtt{X} \mapsto 0\}$

the following is not true (since $\mathtt{X} = \mathtt{Y}$ is a possibility):

$\{\mathtt{X} \mapsto \_\ \wedge\ \mathtt{Y} \mapsto 1\}\ \mathtt{[X]:=}0\ \{\mathtt{X} \mapsto 0\ \wedge\ \mathtt{Y} \mapsto 1\}$

They key insight, attributed to O'Hearn by Reynolds, is to use $\star$ instead of $\wedge$ to ensure that the added assertion R is disjoint from P and Q. This gives rise to the frame rule below:

The frame rule

$$\frac{\vdash\ \{P\}\,C\,\{Q\}}{\vdash\ \{P \star R\}\,C\,\{Q \star R\}}$$

where no variable modified by C occurs free in R.

[2] `http://en.wikipedia.org/wiki/Frame_problem`

In the frame rule a variable V is said to be *modified by* C is it occurs on the left of `:=` in a store, fetch or allocation assignment in C.

The proof that the frame rule is sound is quite tricky and depends on the no-faulting semantics of Hoare triples. The key lemmas are *Monotonicity*:

$$\begin{array}{l}\forall C\ s\ h_0\ h_1\ h_2.\\ \quad\neg(\mathsf{SHsem}\ C\ (s,h_0)\ \mathsf{fault}) \wedge \mathsf{Sep}\,h_0\,h_1\,h_2 \Rightarrow \neg(\mathsf{SHsem}\ C\ (s,h_2)\ \mathsf{fault})\end{array}$$

and *The Frame Property*:

$$\begin{array}{l}\forall C\ s\ s'\ h_0\ h_1\ h_2\ h'.\\ \quad\neg(\mathsf{SHsem}\ C\ (s,h_0)\ \mathsf{fault}) \wedge \mathsf{SHsem}\ C\ (s,h_2)\ (s',h') \wedge \mathsf{Sep}\,h_0\,h_1\,h_2\\ \quad\Rightarrow\\ \quad\exists h_0'.\ \mathsf{SHsem}\ C\ (s,h0)\ (s',h_0') \wedge \mathsf{Sep}\,h_0'\,h_1\,h'\end{array}$$

For further details of what these lemmas mean and why they are key to the soundness of the fame rule see the original paper [25]. Notice that in these two lemmas the quantification is over commands C, not over arbitrary functions $c : \mathit{State} \to \mathit{Result} \to \mathit{Bool}$. This is because the lemmas do not hold for arbitrary functions, only for functions that are the meaning of commands (e.g. for $\mathsf{Csem}\ C$). Abstract separation logic assumes these lemmas as axioms and then develops a generalised version of separation logic that can be instantiated to different models of states. The original paper on abstract separation logic [30] provides more details. See also recent research by Thomas Tuerk [31] on using abstract separation logic as a framework for building mechanised program verification tools.

7.6 Example

The informal Hoare triple:

$$\{\mathit{contents\ of\ pointers}\ \mathtt{X}\ \mathit{and}\ \mathtt{Y}\ \mathit{are\ equal}\}\ \mathtt{X:=[X]\,;\ Y:=[Y]}\ \{\mathtt{X} = \mathtt{Y}\}$$

can be formalised as

$$\{\exists v.\ \mathtt{X} \mapsto v \star \mathtt{Y} \mapsto v\}\ \mathtt{X:=[X]\,;\ Y:=[Y]}\ \{\mathtt{X} = \mathtt{Y}\}$$

By the fetch assignment axiom:

$$\vdash\ \{(\mathtt{X} = x) \wedge \mathtt{X} \mapsto v\}\ \mathtt{X:=[X]}\ \{(\mathtt{X} = v) \wedge x \mapsto v\}$$

$$\vdash\ \{(\mathtt{Y} = y) \wedge \mathtt{Y} \mapsto v\}\ \mathtt{Y:=[Y]}\ \{(\mathtt{Y} = v) \wedge y \mapsto v\}$$

By the frame rule:

$$\begin{array}{ll}\vdash & \{((\mathtt{X}=x)\wedge \mathtt{X}\mapsto v)\star((\mathtt{Y}=y)\wedge \mathtt{Y}\mapsto v)\}\\ & \mathtt{X:=[X]}\\ & \{((\mathtt{X}=v)\wedge x\mapsto v)\star(((\mathtt{Y}=y)\wedge \mathtt{Y}\mapsto v))\}\end{array}$$

$$\begin{array}{ll}\vdash & \{((\mathtt{Y}=y)\wedge \mathtt{Y}\mapsto v)\star((\mathtt{X}=v)\wedge x\mapsto v)\}\\ & \mathtt{Y:=[Y]}\\ & \{((\mathtt{Y}=v)\wedge y\mapsto v)\star((\mathtt{X}=v)\wedge x\mapsto v)\}\end{array}$$

Hence by the sequencing rule and the commutativity of the $\star$ operator (see next section):

$$\begin{array}{ll}\vdash & \{((\mathtt{X}=x)\wedge \mathtt{X}\mapsto v)\star((\mathtt{Y}=y)\wedge \mathtt{Y}\mapsto v)\}\\ & \mathtt{X:=[X];Y:=[Y]}\\ & \{((\mathtt{X}=v)\wedge x\mapsto v)\star((\mathtt{Y}=v)\wedge y\mapsto v)\}\end{array}$$

Next use the exists introduction rule three times to get:

$$\begin{array}{ll}\vdash & \{\exists v\ x\ y.\ ((\mathtt{X}=x)\wedge \mathtt{X}\mapsto v)\star((\mathtt{Y}=y)\wedge \mathtt{Y}\mapsto v)\}\\ & \mathtt{X:=[X];Y:=[Y]}\\ & \{\exists v\ x\ y.\ ((\mathtt{X}=v)\wedge x\mapsto v)\star((\mathtt{Y}=v)\wedge y\mapsto v)\}\end{array}$$

The following implications are true (we say more on why later):

$$(\exists v.\ \mathtt{X}\mapsto v\star\mathtt{Y}\mapsto v)\ \Rightarrow\ \exists v\ x\ y.\ ((\mathtt{X}=x)\wedge \mathtt{X}\mapsto v)\star((\mathtt{Y}=y)\wedge \mathtt{Y}\mapsto v)$$

$$(\exists v\ x\ y.\ ((\mathtt{X}=v)\wedge x\mapsto v)\star((\mathtt{Y}=v)\wedge y\mapsto v))\ \Rightarrow\ (\mathtt{X}=\mathtt{Y})$$

Hence by the rules of consequence:

$$\vdash\ \ \{\exists v.\ \mathtt{X}\mapsto v\star\mathtt{Y}\mapsto v\}\ \mathtt{X:=[X];\ Y:=[Y]}\ \{\mathtt{X}=\mathtt{Y}\}$$

This proof seems rather heavy for such a trivial result, but, as we have seen for simple Hoare logic, derived rules and automation can eliminate most of the fine details. In the next section we say more about proving formulae like the two implications we used with the rules of consequence in the last step.

7.7 The logic of separating assertions

In simple Hoare logic the assertion language consists of standard predicate calculus formulae and thus the standard deductive system of predicate logic can be used to prove formulae, e.g. when needed for applying the rules of consequence. Alternatively one can take a semantic view and regard assertions as predicates on the state and then just use 'ordinary mathematics' to prove assertions.

In separation logic there are additional operators such as $\star$ and $\mapsto$ which are not part of standard logic. One can try to develop a deductive system for such operators and then prove properties of the assertions, but (as far as I know, e.g. [28]) there is no complete deductive system for such assertions. One can accumulate a collection of ad hoc rules for doing proofs, but, as Reynolds says in his notes [23] these are likely to be "far from complete", though they might be good enough for most examples that come up in practice. In the last section it was asserted that the following two implications were true:

$$(\exists v.\ \mathtt{X} \mapsto v \star \mathtt{Y} \mapsto v) \ \Rightarrow\ \exists v\ x\ y.\ ((\mathtt{X} = x) \wedge \mathtt{X} \mapsto v) \star ((\mathtt{Y} = y) \wedge \mathtt{Y} \mapsto v)$$

$$(\exists v\ x\ y.\ ((\mathtt{X} = v) \wedge x \mapsto v) \star ((\mathtt{Y} = v) \wedge y \mapsto v)) \ \Rightarrow\ (\mathtt{X} = \mathtt{Y})$$

To verify that these are true one must show that they hold for all states (s, h) – i.e. that $\forall s\ h.\ \mathsf{SSsem}\ P\ (s, h)$. One could just prove this directly from the definitions, but an alternative is to use derived laws for the separation logic operators to prove the assertions 'algebraically'. For example, the following equations can be derived from the definition of $\star$ (see Section 7.3.2):

$$\exists x.\ P_1 \star P_2 = P_1 \star (\exists x.\ P_2) \qquad \text{(when } x \text{ not free in } P_1\text{)}$$

$$\exists x.\ P_1 \star P_2 = (\exists x.\ P_1) \star P_2 \qquad \text{(when } x \text{ not free in } P_2\text{)}$$

hence:

$$\begin{aligned}
&\exists v\ x\ y.\ ((\mathtt{X} = x) \wedge \mathtt{X} \mapsto v) \star ((\mathtt{Y} = y) \wedge \mathtt{Y} \mapsto v)\\
&\quad = \ \exists v.\ (\exists x.\ (\mathtt{X} = x) \wedge \mathtt{X} \mapsto v) \star (\exists y.\ (\mathtt{Y} = y) \wedge \mathtt{Y} \mapsto v)\\
&\quad = \ \exists v.\ ((\exists x.\ \mathtt{X} = x) \wedge \mathtt{X} \mapsto v) \star ((\exists y.\ \mathtt{Y} = y) \wedge \mathtt{Y} \mapsto v)\\
&\quad = \ \exists v.\ (\mathtt{T} \wedge \mathtt{X} \mapsto v) \star (\mathtt{T} \wedge \mathtt{Y} \mapsto v)\\
&\quad = \ \exists v.\ \mathtt{X} \mapsto v \star \mathtt{Y} \mapsto v
\end{aligned}$$

This establishes the first implication (actually it establishes a stronger result: an equation rather than an implication).

To prove the second implication, first start by a similar calculation to the one above:

$$\begin{aligned}
&(\exists v\ x\ y.\ ((\mathtt{X} = v) \wedge x \mapsto v) \star ((\mathtt{Y} = v) \wedge y \mapsto v))\\
&\quad = \ \exists v.\ ((\mathtt{X} = v) \wedge (\exists x.\ x \mapsto v)) \star ((\mathtt{Y} = v) \wedge (\exists y.\ y \mapsto v))
\end{aligned}$$

We say a property is *heap independent* if it doesn't depend on the heap. The classical statements discussed in Section 7.3 are heap independent. Semantically P is heap independent iff $\forall s\ h_1\ h_2.\ P(s, h_1) = P(s, h_2)$. The following law is then true:

$$((P_1 \wedge Q_1) \star (P_2 \wedge Q_2)) \Rightarrow (P_1 \wedge P_2) \qquad (P_1,\ P_2 \text{ heap independent})$$

The values of variables don't depend on the heap, so both $\texttt{X} = v$ and $\texttt{Y} = v$ are heap independent. Thus:

$$
\begin{aligned}
&(\exists v.\ ((\texttt{X} = v) \wedge (\exists x.\ x \mapsto v)) \star ((\texttt{Y} = v) \wedge (\exists y.\ y \mapsto v)))\\
&\ \Rightarrow \exists v.\ (\texttt{X} = v) \wedge (\texttt{Y} = v)\\
&\ \Rightarrow \texttt{X} = \texttt{Y}
\end{aligned}
$$

This completes the proof that:

$$(\exists v\ x\ y.\ ((\texttt{X} = v) \wedge x \mapsto v) \star ((\texttt{Y} = v) \wedge y \mapsto v)) \ \Rightarrow\ (\texttt{X} = \texttt{Y})$$

7.8 The list reversal program

In this section we take a preliminary look at the list reversing program discussed earlier. Further details (e.g. a full proof) may be added to a future version of these notes. A proof outline can be found in Reynolds notes [23].

The Hoare triple to be proved is:

```
{list α₀ X}
Y:=nil;
WHILE ¬(X = nil) DO (Z:=[X+1]; [X+1]:=Y; Y:=X; X:=Z)
{list (rev(α₀)) Y}
```

We previously mentioned that "·" is the list concatenation operator (we will also write $a \cdot \alpha$ for the result of 'consing' an element a onto α). The invariant given by Reynolds in his notes is:

$$\exists \alpha\ \beta.\ \mathsf{list}\ \alpha\ \texttt{X} \star \mathsf{list}\ \beta\ \texttt{Y}\ \wedge\ (\mathsf{rev}(\alpha_0) = \mathsf{rev}(\alpha) \cdot \beta)$$

We need to show that:

1. this holds just before the loop is entered;
2. it is indeed an invariant;
3. with the loop exit condition $\texttt{X} = \mathsf{nil}$ it implies $\mathsf{list}\ (\mathsf{rev}(\alpha_0))\ \texttt{Y}$.

What follows has not been fully checked and may contain errors!

To show 1 we need to prove:

$$\{\mathsf{list}\ \alpha_0\ \texttt{X}\}\ \texttt{Y:=}\mathsf{nil}\ \{\exists \alpha\ \beta.\ \mathsf{list}\ \alpha\ \texttt{X} \star \mathsf{list}\ \beta\ \texttt{Y}\ \wedge\ (\mathsf{rev}(\alpha_0) = \mathsf{rev}(\alpha) \cdot \beta)\}$$

By the store assignment axiom:

$$\vdash\ \{\mathtt{Y} \doteq v\}\ \mathtt{Y\!:=}\mathsf{nil}\ \{\mathtt{Y} \doteq \mathsf{nil}[v/\mathtt{Y}]\}$$

hence, as $\mathtt{Y}$ doesn't occur in nil:

$$\vdash\ \{\mathtt{Y} \doteq v\}\ \mathtt{Y\!:=}\mathsf{nil}\ \{\mathtt{Y} \doteq \mathsf{nil}\}$$

By the definition of list (base case): $\mathsf{list}\ \mathtt{[]}\ e\ =\ (e \doteq \mathsf{nil})$

$$\vdash\ \{\mathtt{Y} \doteq v\}\ \mathtt{Y\!:=}\mathsf{nil}\ \{\mathsf{list}\ \mathtt{[]}\ \mathtt{Y}\}$$

By the frame rule (and commutativity of $\star$):

$$\vdash\ \{\mathsf{list}\ \alpha_0\ \mathtt{X} \star (\mathtt{Y} \doteq v)\}\ \mathtt{Y\!:=}\mathsf{nil}\ \{\mathsf{list}\ \alpha_0\ \mathtt{X} \star \mathsf{list}\ \mathtt{[]}\ \mathtt{Y}\}$$

Clearly $\mathsf{rev}(\alpha_0) = \mathsf{rev}(\alpha_0) \cdot \mathtt{[]}$, so:

$$\vdash\ \{\mathsf{list}\ \alpha_0\ \mathtt{X} \star (\mathtt{Y} \doteq v)\}\ \mathtt{Y\!:=}\mathsf{nil}\ \{\mathsf{list}\ \alpha_0\ \mathtt{X} \star \mathsf{list}\ \mathtt{[]}\ \mathtt{Y} \wedge (\mathsf{rev}(\alpha_0) = \mathsf{rev}(\alpha_0) \cdot \mathtt{[]})\}$$

By exists introduction (see Section 7.4.1):

$$\vdash\ \{\exists v.\ \mathsf{list}\ \alpha_0\ \mathtt{X} \star (\mathtt{Y} \doteq v)\}\ \mathtt{Y\!:=}\mathsf{nil}\ \{\exists v.\ \mathsf{list}\ \alpha_0\ \mathtt{X} \star \mathsf{list}\ \mathtt{[]}\ \mathtt{Y} \wedge (\mathsf{rev}(\alpha_0) = \mathsf{rev}(\alpha_0) \cdot \mathtt{[]})\}$$

Let us assume the following two purely logical implications:

$$(P1.1)\quad \vdash\ \mathsf{list}\ \alpha_0\ \mathtt{X} \Rightarrow \exists v.\ \mathsf{list}\ \alpha_0\ \mathtt{X} \star (\mathtt{Y} \doteq v)$$

$$\begin{array}{ll} (P1.2)\quad \vdash & (\exists v.\ \mathsf{list}\ \alpha_0\ \mathtt{X} \star \mathsf{list}\ \mathtt{[]}\ \mathtt{Y} \wedge (\mathsf{rev}(\alpha_0) = \mathsf{rev}(\alpha_0) \cdot \mathtt{[]})) \\ & \Rightarrow \\ & (\exists \alpha\ \beta.\ \mathsf{list}\ \alpha\ \mathtt{X} \star \mathsf{list}\ \beta\ \mathtt{Y}\ \wedge\ (\mathsf{rev}(\alpha_0) = \mathsf{rev}(\alpha) \cdot \beta)) \end{array}$$

From $P1.1$ and $P1.2$, the result of exists introduction above and the consequence rules:

$$\{\mathsf{list}\ \alpha_0\ \mathtt{X}\}\ \mathtt{Y\!:=}\mathsf{nil}\ \{\exists \alpha\ \beta.\ \mathsf{list}\ \alpha\ \mathtt{X} \star \mathsf{list}\ \beta\ \mathtt{Y}\ \wedge\ (\mathsf{rev}(\alpha_0) = \mathsf{rev}(\alpha) \cdot \beta)\}$$

which is 1.

To show 2 we need to prove:

$$\begin{array}{l} \{(\exists \alpha\ \beta.\ \mathsf{list}\ \alpha\ \mathtt{X} \star \mathsf{list}\ \beta\ \mathtt{Y}\ \wedge\ (\mathsf{rev}(\alpha_0) = \mathsf{rev}(\alpha) \cdot \beta)) \wedge \neg(\mathtt{X} = \mathsf{nil})\} \\ \mathtt{Z\!:=\![X+1]};\ \mathtt{[X+1]\!:=\!Y};\ \mathtt{Y\!:=\!X};\ \mathtt{X\!:=\!Z} \\ \{\exists \alpha\ \beta.\ \mathsf{list}\ \alpha\ \mathtt{X} \star \mathsf{list}\ \beta\ \mathtt{Y}\ \wedge\ (\mathsf{rev}(\alpha_0) = \mathsf{rev}(\alpha) \cdot \beta)\} \end{array}$$

which we do by proving the following three statements and then using the Sequencing Rule.

$$\begin{array}{l} \{(\exists \alpha\ \beta.\ \mathsf{list}\ \alpha\ \mathtt{X} \star \mathsf{list}\ \beta\ \mathtt{Y}\ \wedge\ (\mathsf{rev}(\alpha_0) = \mathsf{rev}(\alpha) \cdot \beta)) \wedge \neg(\mathtt{X} = \mathsf{nil})\} \\ \mathtt{Z\!:=\![X+1]} \\ \{\exists a\ \alpha\ \beta.\ \mathtt{X} \mapsto a, \mathtt{Z} \star \mathsf{list}\ \alpha\ \mathtt{Z} \star \mathsf{list}\ \beta\ \mathtt{Y}\ \wedge\ (\mathsf{rev}(\alpha_0) = \mathsf{rev}(a \cdot \alpha) \cdot \beta)\} \end{array}$$

$$\begin{array}{l} \{\exists a\ \alpha\ \beta.\ \mathtt{X} \mapsto a, \mathtt{Z} \star \mathsf{list}\ \alpha\ \mathtt{Z} \star \mathsf{list}\ \beta\ \mathtt{Y}\ \wedge\ (\mathsf{rev}(\alpha_0) = \mathsf{rev}(a \cdot \alpha) \cdot \beta)\} \\ \mathtt{[X+1]\!:=\!Y} \\ \{\exists \alpha\ \beta.\ \mathsf{list}\ \alpha\ \mathtt{Z} \star \mathsf{list}\ \beta\ \mathtt{X}\ \wedge\ (\mathsf{rev}(\alpha_0) = \mathsf{rev}(\alpha) \cdot \beta)\} \end{array}$$

$$
\begin{array}{l}
\{\exists\alpha\ \beta.\ \mathsf{list}\ \alpha\ \mathtt{Z} \star \mathsf{list}\ \beta\ \mathtt{X}\ \wedge\ (\mathsf{rev}(\alpha_0) = \mathsf{rev}(\alpha)\cdot\beta)\}\\
\mathtt{Y:=X;X:=Z}\\
\{\exists\alpha\ \beta.\ \mathsf{list}\ \alpha\ \mathtt{X} \star \mathsf{list}\ \beta\ \mathtt{Y}\ \wedge\ (\mathsf{rev}(\alpha_0) = \mathsf{rev}(\alpha)\cdot\beta)\}
\end{array}
$$

The last of these follows by two applications of the ordinary Hoare assignment axiom and the sequencing rule. The first two are more tricky, and require the fetch and heap assignment axioms, respectively. Recall:

Fetch assignment axiom

$$\vdash\ \{(V = v_1) \wedge E \mapsto v_2\}\ V\mathtt{:=[}E\mathtt{]}\ \{(V = v_2) \wedge E\mathtt{[}v_1\mathtt{/}V\mathtt{]} \mapsto v_2\}$$

where v_1, v_2 are auxiliary variables not occurring in E.

The instance of this we need is:

$$\vdash\ \{(\mathtt{Z} = v_1) \wedge \mathtt{X+1} \mapsto v_2\}\ \mathtt{Z:=[X+1]}\ \{(\mathtt{Z} = v_2) \wedge \mathtt{X+1[}v_1\mathtt{/Z]} \mapsto v_2\}$$

As $\mathtt{Z}$ does not occur in $\mathtt{X+1}$ we have $\mathtt{X+1[}v_1\mathtt{/Z]} = \mathtt{X+1}$. The variable v_1 serves no useful role here, so we can eliminate it by instantiating it to $\mathtt{Z}$. We also rename the logical variable v_2 to l. Thus:

$$\vdash\ \{\mathtt{X+1} \mapsto l\}\ \mathtt{Z:=[X+1]}\ \{(\mathtt{Z} = l) \wedge \mathtt{X+1} \mapsto l\}$$

This is a local property just describing the change to a one-element heap (containing $\mathtt{X+1}$). From this, we must somehow deduce a global property about the whole list. Let :

$$R\ =\ \mathtt{X} \mapsto a \star \mathsf{list}\ \alpha'\ l \star \mathsf{list}\ \beta\ \mathtt{Y} \wedge (\mathsf{rev}(\alpha_0) = \mathsf{rev}(a\cdot\alpha')\cdot\beta) \wedge \neg(\mathtt{X} = \mathsf{nil})$$

The process of finding this R is related to *abduction*, a kind of frame inference that is a hot topic in recent research [4]. By the frame rule, followed by repeated applications of the exists rule:

$$\vdash\ \{\exists\alpha\ \beta\ a\ l\ \alpha'.\ \mathtt{X+1} \mapsto l \star R\}\ \mathtt{Z:=[X+1]}\ \{\exists\alpha\ \beta\ a\ l\ \alpha'.\ (\mathtt{Z} = l) \wedge \mathtt{X+1} \mapsto l \star R\}$$

From this we need to deduce:

$$
\begin{array}{ll}
\vdash & \{(\exists\alpha\ \beta.\ \mathsf{list}\ \alpha\ \mathtt{X} \star \mathsf{list}\ \beta\ \mathtt{Y}\ \wedge\ (\mathsf{rev}(\alpha_0) = \mathsf{rev}(\alpha)\cdot\beta)) \wedge \neg(\mathtt{X} = \mathsf{nil})\}\\
& \mathtt{Z:=[X+1]}\\
& \{\exists a\ \alpha\ \beta.\ \mathtt{X} \mapsto a, \mathtt{Z} \star \mathsf{list}\ \alpha\ \mathtt{Z} \star \mathsf{list}\ \beta\ \mathtt{Y}\ \wedge\ (\mathsf{rev}(\alpha_0) = \mathsf{rev}(a\cdot\alpha)\cdot\beta)\}
\end{array}
$$

which can be done using the consequence rules if $P2.1$ and $P2.2$ below hold:

$$
\begin{array}{lll}
(P2.1) & \vdash & (\exists\alpha\ \beta.\ (\mathsf{list}\ \alpha\ \mathtt{X} \star \mathsf{list}\ \beta\ \mathtt{Y}\ \wedge\ (\mathsf{rev}(\alpha_0) = \mathsf{rev}(\alpha)\cdot\beta)) \wedge \neg(\mathtt{X} = \mathsf{nil}))\\
& & \Rightarrow \exists\alpha\ \beta\ a\ l\ \alpha'.\ (\mathtt{X+1} \mapsto l) \star R\\
(P2.2) & \vdash & (\exists\alpha\ \beta\ a\ l\ \alpha'.\ ((\mathtt{Z} = l) \wedge \mathtt{X+1} \mapsto l) \star R)\\
& & \Rightarrow \exists a\ \alpha\ \beta.\ \mathtt{X} \mapsto a, \mathtt{Z} \star \mathsf{list}\ \alpha\ \mathtt{Z} \star \mathsf{list}\ \beta\ \mathtt{Y}\ \wedge\ (\mathsf{rev}(\alpha_0) = \mathsf{rev}(a\cdot\alpha)\cdot\beta)
\end{array}
$$

These are purely logical properties (in the assertion language of separation logic). Their proof uses the definition of the list predicate list and logical reasoning. Recall the list predicate:

$$\boxed{\begin{array}{l} \mathsf{list}\ \texttt{[]}\ e\ =\ (e \doteq \mathsf{nil}) \\ \mathsf{list}\ ([a_0, a_1, \ldots, a_n])\ e\ =\ \exists e'.\ (e \mapsto a_0, e')\ \star\ \mathsf{list}\ [a_1, \ldots, a_n]\ e' \end{array}}$$

where

$$\boxed{E \mapsto F_0, \ldots, F_n\ =\ (E \mapsto F_0) \star \cdots \star (E{+}n \mapsto F_n)}$$

so

$$\boxed{\begin{array}{l} \mathsf{list}\ \texttt{[]}\ e = (e \doteq \mathsf{nil}) \\ \mathsf{list}\ ([a_0, a_1, \ldots, a_n])\ e = \exists e'.\ (e \mapsto a_0) \star (e{+}1 \mapsto e')\ \star\ \mathsf{list}\ [a_1, \ldots, a_n]\ e' \end{array}}$$

Arguing informally: from $\mathsf{list}\ \alpha\ \texttt{X}$ and $\neg(\texttt{X} = \mathsf{nil})$ it follows that for some value a and α' we have $\alpha = a \cdot \alpha'$. From this $\mathsf{rev}(\alpha_0) = \mathsf{rev}(a \cdot \alpha') \cdot \beta$ and from $\mathsf{list}\ \alpha\ \texttt{X}$ there exists a location l such that $\texttt{X} \mapsto a$, $\texttt{X+1} \mapsto l$ and $\mathsf{list}\ \alpha'\ l$. Thus:

$$\begin{array}{ll} \vdash & (\mathsf{list}\ \alpha\ \texttt{X} \star \mathsf{list}\ \beta\ \texttt{Y}\ \wedge\ (\mathsf{rev}(\alpha_0) = \mathsf{rev}(\alpha) \cdot \beta)) \wedge \neg(\texttt{X} = \mathsf{nil}) \\ & \Rightarrow \\ & ((\exists a\ l\ \alpha'. \\ & \quad \texttt{X} \mapsto a \star \texttt{X+1} \mapsto l \star \mathsf{list}\ \alpha'\ l \star \mathsf{list}\ \beta\ \texttt{Y} \\ & \quad \wedge\ (\mathsf{rev}(\alpha_0) = \mathsf{rev}(a \cdot \alpha') \cdot \beta))\ \wedge\ \neg(\texttt{X} = \mathsf{nil})) \end{array}$$

The first of the two needed logical properties follows from this using some quantifier movement and the commutativity of $\star$. The second property requires the list predicate to be unfolded.

This concludes a sketch of the proof of the first Hoare triple:

$$\begin{array}{l} \{(\exists \alpha\ \beta.\ \mathsf{list}\ \alpha\ \texttt{X} \star \mathsf{list}\ \beta\ \texttt{Y}\ \wedge\ (\mathsf{rev}(\alpha_0) = \mathsf{rev}(\alpha) \cdot \beta)) \wedge \neg(\texttt{X} = \mathsf{nil})\} \\ \texttt{Z:=[X+1]} \\ \{\exists a\ \alpha\ \beta.\ \texttt{X} \mapsto a, \texttt{Z} \star \mathsf{list}\ \alpha\ \texttt{Z} \star \mathsf{list}\ \beta\ \texttt{Y}\ \wedge\ (\mathsf{rev}(\alpha_0) = \mathsf{rev}(a \cdot \alpha) \cdot \beta)\} \end{array}$$

The remaining Hoare triple is:

$$\begin{array}{l} \{\exists a\ \alpha\ \beta.\ \texttt{X} \mapsto a, \texttt{Z} \star \mathsf{list}\ \alpha\ \texttt{Z} \star \mathsf{list}\ \beta\ \texttt{Y}\ \wedge\ (\mathsf{rev}(\alpha_0) = \mathsf{rev}(a \cdot \alpha) \cdot \beta)\} \\ \texttt{[X+1]:=Y} \\ \{\exists \alpha\ \beta.\ \mathsf{list}\ \alpha\ \texttt{Z} \star \mathsf{list}\ \beta\ \texttt{X}\ \wedge\ (\mathsf{rev}(\alpha_0) = \mathsf{rev}(\alpha) \cdot \beta)\} \end{array}$$

To prove this we need the heap assignment axiom:

Heap assignment axiom

$$\vdash\ \{E \mapsto \_\}\ \texttt{[}E\texttt{]:=}F\ \{E \mapsto F\}$$

The appropriate instance is:

$$\vdash\ \{\exists v.\ \texttt{X+1} \mapsto v\}\ \texttt{[X+1]:=Y}\ \{\texttt{X+1} \mapsto \texttt{Y}\}$$

By inventing a suitable frame, application of the frame rule and some logical fiddling, including using the definition of list, one deduces from this:

$$\begin{array}{l}\vdash\ \{\exists a\ \alpha\ \beta.\ \texttt{X} \mapsto a, \texttt{Z} \star \mathsf{list}\ \alpha\ \texttt{Z} \star \mathsf{list}\ \beta\ \texttt{Y}\ \wedge\ (\mathsf{rev}(\alpha_0) = \mathsf{rev}(a \cdot \alpha) \cdot \beta)\}\\ \quad \texttt{[X+1]:=Y}\\ \quad \{\exists a\ \alpha\ \beta.\ \texttt{X} \mapsto a, \texttt{Y} \star \mathsf{list}\ \alpha\ \texttt{Z} \star \mathsf{list}\ \beta\ \texttt{Y}\ \wedge\ (\mathsf{rev}(\alpha_0) = \mathsf{rev}(a \cdot \alpha) \cdot \beta)\}\end{array}$$

and then one gets the desired result by postcondition weakening using:

$$\begin{array}{lll}(P2.3) & \vdash & (\exists a\ \alpha\ \beta.\ \texttt{X} \mapsto a, \texttt{Y} \star \mathsf{list}\ \alpha\ \texttt{Z} \star \mathsf{list}\ \beta\ \texttt{Y}\ \wedge\ (\mathsf{rev}(\alpha_0) = \mathsf{rev}(a \cdot \alpha) \cdot \beta))\\ & & \Rightarrow\\ & & (\exists \alpha\ \beta.\ \mathsf{list}\ \alpha\ \texttt{Z} \star \mathsf{list}\ \beta\ \texttt{X}\ \wedge\ (\mathsf{rev}(\alpha_0) = \mathsf{rev}(\alpha) \cdot \beta))\end{array}$$

which is proved by first proving:

$$\begin{array}{lll}(P2.3.1) & \vdash & (\exists a\ \alpha\ \beta.\ \texttt{X} \mapsto a, \texttt{Y} \star \mathsf{list}\ \alpha\ \texttt{Z} \star \mathsf{list}\ \beta\ \texttt{Y}\ \wedge\ (\mathsf{rev}(\alpha_0) = \mathsf{rev}(a \cdot \alpha) \cdot \beta))\\ & & \Rightarrow\\ & & (\exists a\ \alpha\ \beta.\ \mathsf{list}\ \alpha\ \texttt{Z} \star \mathsf{list}\ (a \cdot \beta)\ \texttt{X}\ \wedge\ (\mathsf{rev}(\alpha_0) = \mathsf{rev}(\alpha) \cdot a \cdot \beta))\end{array}$$

and then proving

$$\begin{array}{lll}(P2.3.2) & \vdash & (\exists a\ \alpha\ \beta.\ \mathsf{list}\ \alpha\ \texttt{Z} \star \mathsf{list}\ (a \cdot \beta)\ \texttt{X}\ \wedge\ (\mathsf{rev}(\alpha_0) = \mathsf{rev}(\alpha) \cdot a \cdot \beta))\\ & & \Rightarrow\\ & & (\exists \alpha\ \beta.\ \mathsf{list}\ \alpha\ \texttt{Z} \star \mathsf{list}\ \beta\ \texttt{X}\ \wedge\ (\mathsf{rev}(\alpha_0) = \mathsf{rev}(\alpha) \cdot \beta))\end{array}$$

and then using the transitivity of implication ($\Rightarrow$).

Finally, to show 3 (i.e. invariant and loop exit condition $\texttt{X}=\mathsf{nil}$ implies $\mathsf{list}\ (\mathsf{rev}(\alpha_0))\ \texttt{Y}$) we need to prove property $P3$, where:

$$\begin{array}{lll}(P3) & \vdash & (\exists \alpha\ \beta.\ \mathsf{list}\ \alpha\ \texttt{X} \star \mathsf{list}\ \beta\ \texttt{Y}\ \wedge\ (\mathsf{rev}(\alpha_0) = \mathsf{rev}(\alpha) \cdot \beta)) \wedge (\texttt{X} = \mathsf{nil})\\ & & \Rightarrow\\ & & \mathsf{list}\ (\mathsf{rev}(\alpha_0))\ \texttt{Y}\end{array}$$

Which, again, is fiddly pure logic using the definition of the list predicate list.

Proofs like the one sketched above, are normally shown as 'proof outlines' which are a similar to annotated programs. Reynolds' proof outline for the list reversing example [23] is (with some renaming of variables and other minor changes):

```
$\{\mathsf{list}\ \alpha_0\ \mathtt{X}\}$
Y:=nil;
$\{\exists\alpha\ \beta.\ \mathsf{list}\ \alpha\ \mathtt{X} \star \mathsf{list}\ \beta\ \mathtt{Y}\ \wedge\ (\mathsf{rev}(\alpha_0) = \mathsf{rev}(\alpha)\cdot\beta)\}$
WHILE $\neg$(X=nil) DO $\{\exists\alpha\ \beta.\ \mathsf{list}\ \alpha\ \mathtt{X} \star \mathsf{list}\ \beta\ \mathtt{Y}\ \wedge\ (\mathsf{rev}(\alpha_0)=\mathsf{rev}(\alpha)\cdot\beta)\}$
$(\{\exists\alpha\ \beta.\ \mathsf{list}\ \alpha\ \mathtt{X} \star \mathsf{list}\ \beta\ \mathtt{Y}\ \wedge\ (\mathsf{rev}(\alpha_0)=\mathsf{rev}(\alpha)\cdot\beta) \wedge \neg(\mathtt{X}=\mathsf{nil})\}$
 $\{\exists\alpha\ \beta\ a\ l\ \alpha'.$
   $(\mathtt{X+1} \mapsto l) \star \mathtt{X} \mapsto a \star \mathsf{list}\ \alpha'\ l \star \mathsf{list}\ \beta\ \mathtt{Y} \wedge (\mathsf{rev}(\alpha_0)=\mathsf{rev}(a\cdot\alpha')\cdot\beta) \wedge \neg(\mathtt{X}=\mathsf{nil})\}$
 Z:=[X+1];
 $\{\exists\alpha\ \beta\ a\ l\ \alpha'.$
   $((\mathtt{Z}=l) \wedge \mathtt{X+1} \mapsto l) \star \mathtt{X} \mapsto a \star \mathsf{list}\ \alpha'\ l \star \mathsf{list}\ \beta\ \mathtt{Y} \wedge (\mathsf{rev}(\alpha_0)=\mathsf{rev}(a\cdot\alpha')\cdot\beta) \wedge \neg(\mathtt{X}=\mathsf{nil})\}$
 $\{\exists a\ \alpha\ \beta.\ \mathtt{X} \mapsto a, \mathtt{Z} \star \mathsf{list}\ \alpha\ \mathtt{Z} \star \mathsf{list}\ \beta\ \mathtt{Y}\ \wedge\ (\mathsf{rev}(\alpha_0) = \mathsf{rev}(a\cdot\alpha)\cdot\beta)\}$
 [X+1]:=Y;
 $\{\exists a\ \alpha\ \beta.\ \mathtt{X} \mapsto a, \mathtt{Y} \star \mathsf{list}\ \alpha\ \mathtt{Z} \star \mathsf{list}\ \beta\ \mathtt{Y}\ \wedge\ (\mathsf{rev}(\alpha_0) = \mathsf{rev}(a\cdot\alpha)\cdot\beta)\}$
 $\{\exists a\ \alpha\ \beta.\ \mathsf{list}\ \alpha\ \mathtt{Z} \star \mathsf{list}\ (a\cdot\beta)\ \mathtt{X}\ \wedge\ (\mathsf{rev}(\alpha_0) = \mathsf{rev}(\alpha)\cdot a\cdot\beta)\}$
 $\{\exists\alpha\ \beta.\ \mathsf{list}\ \alpha\ \mathtt{Z} \star \mathsf{list}\ \beta\ \mathtt{X}\ \wedge\ (\mathsf{rev}(\alpha_0) = \mathsf{rev}(\alpha)\cdot\beta)\}$
 Y:=X; X:=Z
 $\{\exists\alpha\ \beta.\ \mathsf{list}\ \alpha\ \mathtt{X} \star \mathsf{list}\ \beta\ \mathtt{Y}\ \wedge\ (\mathsf{rev}(\alpha_0) = \mathsf{rev}(\alpha)\cdot\beta)\})$
$\{\mathsf{list}\ (\mathsf{rev}(\alpha_0))\ \mathtt{Y}\}$
```

Proof outlines like this are superficially similar to annotated Hoare triples as described for verification condition generation. They do specify what has to be done to get a complete proof, namely:

- prove $\vdash\ P \Rightarrow Q$ for each sequence of sentences $\{P\}\{Q\}$;
- prove $\vdash\ \{P\}\ C\ \{Q\}$ for each occurrence of a Hoare triple.

However, proving these is not always straightforward or mechanisable.

- There is no established methodology for proving $P \Rightarrow Q$ when P, Q are arbitrary assertions of separation logic – one relies on manual methods from incomplete sets of axioms and rules, or decision procedures for weak subsets.
- The assignment axioms of separation logic only support local reasoning about the sub-heaps involved - one needs to the extend local Hoare triples given by the axioms to global ones using the frame rule, and finding the right frame to use is tricky and heuristic, somewhat analogous to finding invariants, rather than algorithmic (it's related to abduction [4]).

Thus proof outlines are (currently) mainly an informal notation for writing down hand proofs.

Mechanising separation logic is an active research area. Most success so far has been on just verifying shape properties (i.e. shape analysis). The classic work is a tool called *Smallfoot* (google `Smallfoot Berdine`). A recent project at Cambridge to mechanise reasoning about the content of data-structures, rather than just their shape, is *Holfoot* (google `Holfoot Tuerk`).

In addition to the mechanisation of separation logic, there is much current research on extending the logic to support mainstream programming methods, like concurrency and object-oriented programing.

Bibliography

[1] Apt, K.R., ‘Ten Years of Hoare’s Logic: A Survey – Part I‘, ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 3, Issue 4, 1981.

[2] Alagić, S. and Arbib, M.A., *The Design of Well-structured and Correct Programs*, Springer-Verlag, 1978.

[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] Calcano, C., Distefano, D., O’Hearn, P. W. and Yang, H., ‘Compositional Shape Analysis by means of Bi-Abduction, JACM (to appear). `www.doc.ic.ac.uk/~ccris/ftp/jacm-abduction.pdf`.

[5] 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.

[6] Cook, S. ‘Soundness and completeness for an axiom system for program verification’. SIAM J. Computing 7, pp. 70-90. 1978.

[7] Dijkstra, E.W., ‘Guarded commands, non-determinacy and formal derivation of programs’, Commun. ACM 18, 1975

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

[9] 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.

[10] Nagel, E. and Newman, J.R., *Gödel’s Proof*, Routledge & Kegan Paul, London, 1959.

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

[12] Gordon, M.J.C., 'Forward with Hoare', in *Reflections on the Work of C. A. R. Hoare*, edited by Jones C. B., Roscoe A. W. and Wood K. R., Springer, 2010.

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

[14] Hoare, C.A.R., 'A Note on the FOR Statement', BIT, **12**, pp. 334-341, 1972.

[15] Joyce, E., 'Software bugs: a matter of life and liability', *Datamation*, **33**, No. 10, May 15, 1987.

[16] Leivant, D. and Fernando, T., 'Skinny and fleshy failures of relative completeness', Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pp. 246-252, 1987.

[17] 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.

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

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

[20] Morgan, C.C., *Programming from Specifications*, Prentice-Hall, 1990.

[21] Morris, J.M. *A Theoretical Basis for Stepwise Refinement and the Programming Calculus*, in *Science of Computer Programming*, vol. 9, pp 287–306, 1989.

[22] Reynolds, J.C., *The Craft of Programming*, Prentice Hall, London, 1981.

[23] Reynolds, J.C., 'Introduction to Separation Logic', unpublished course notes available online at `http://www.cs.cmu.edu/~jcr/`, CMU, 2011.

[24] Winskel, G., *The Formal Semantics of Programming Languages*, MIT Press, 1993.

[25] Yang, H. and O'Hearn, P. W., 'A Semantic Basis for Local Reasoning', in *Proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures*, Springer-Verlag, 2002.

[26] O'Hearn, P., Reynolds, J.C. and Yang, H., 'Local reasoning about programs that alter data structures'. In *Proceedings of 15th Annual Conference of the European Association for Computer Science Logic: CSL 2001*, edited by Fribourg, L., LNCS 2142, Springer-Verlag, 2001.

[27] Burstall, R. M., 'Some techniques for proving correctness of programs which alter data structures', in *Machine Intelligence 7* eds. Meltzer, B. and Michie, D., Edinburgh University Press, 1972.

[28] Calcagno, C., Yang, H. and O'Hearn, P. W., 'Computability and Complexity Results for a Spatial Assertion Language for Data Structures'. In *Proceedings of the 21st Conference on Foundations of Software Technology and Theoretical Computer Science*, ISBN: 3-540-43002-4, Springer-Verlag, 2001.

[29] Stoy, J. E., *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, 1977.

[30] Calcagno, C., O'Hearn, P. W. and Yang, H., 'Local Action and Abstract Separation Logic'. In LICS '07: Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science (2007).

[31] Tuerk, T., 'A Separation Logic Framework in HOL'. In *Theorem Proving in Higher Order Logics: Emerging Trends Proceedings*, Department of Electrical Computer Engineering, Concordia University, 2008. `http://www.cl.cam.ac.uk/~tt291/publications/Tuer08.pdf`.