

## 2 Formal Specification

### 2.1 Introduction

#### Formal Specification

- A formal specification:
  - describes a program's effect
  - in a mathematically precise notation
- Formal specifications:
  - force the specifier to think more precisely about what the specification says
  - provide less scope for confusion about the meaning of the specification
  - can be used to prove the correctness of programs
  - can be used to generate test cases
- Our formal notation:
  - is based on predicate calculus
  - applies to *imperative* programs
  - is due to Tony Hoare of Oxford and Microsoft

#### Formal Specification Languages

**model-based** A set of primitive mathematical domains is assumed. Operations use (maybe implicitly) a couple of predicates to establish the connection between input and output data: pre- and post-conditions. Some examples: Z, VDM, Larch, B, etc.

**algebraic** These allow for self-contained specifications independent of the representation of data. The system is decomposed into a set of abstract data types and each operation specified in terms of its relation with the others, possibly by means of an equational theory. Some examples: Clear, OBJ, ACT-ONE, etc.

### 2.2 Specification of Imperative Programs

#### Imperative Languages

- Executing an imperative program has the effect of changing the *state*
  - i.e. the values of program variables
  - N.B. languages more complex than those described here may have states consisting of other things than the values of variables (e.g. files, I/O)
- To use an imperative program
  - first establish an initial state
  - i.e. set some variables to have values of interest
  - then execute the program (to transform the initial state into a final one)
- One then inspects the values of variables in the final state to get the desired results

## A Small Imperative Programming Language

Summary of syntax:

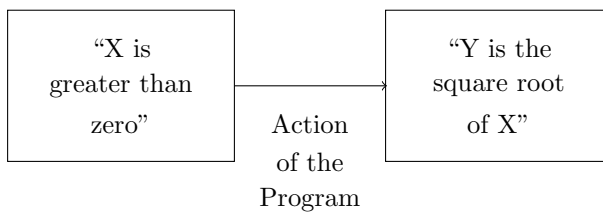
```

 $E ::= N \mid V \mid E_1 + E_2 \mid E_1 - E_2 \mid E_1 \times E_2 \mid \dots$ 
 $B ::= T \mid F \mid E_1 = E_2 \mid E_1 \leq E_2 \mid \dots$ 
 $C ::= \text{SKIP}$ 
      |  $V := E$ 
      |  $C_1 ; C_2$ 
      | IF  $B$  THEN  $C_1$  ELSE  $C_2$ 
      | BEGIN VAR  $V_1 ; \dots \text{VAR } V_n ; C$  END
      | WHILE  $B$  DO  $C$ 

```

## Specification of Imperative Programs

Acceptable Initial State	Acceptable Final State
-----------------------------	---------------------------



## 2.3 Floyd-Hoare Logic

### Partial Correctness

- Tony Hoare introduced the notation  $\{P\} C \{Q\}$ , called a *partial correctness specification*, for specifying what a program does, where:
  - $C$  is a program from the programming language whose programs are being specified
  - $P$  and  $Q$  are conditions on the program variables in  $C$
  - $P$  is called its *precondition*
  - $Q$  its *postcondition*
- $\{P\} C \{Q\}$  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  $Q$
- These specifications are 'partial' because for  $\{P\} C \{Q\}$  to be true it is *not* necessary for the execution of  $C$  to terminate when started in a state satisfying  $P$
- It is only required that *if* the execution terminates, *then*  $Q$  holds

## Specifications as the Basis of Contracts

- A formal specification can be used as the basis for requirements analysis and procurement
  - is this what the customer intended?
  - *design by contract*
- “I want a program that swaps the values in X and Y”
- $\{X = x \wedge Y = y\} C \{X = y \wedge Y = x\}$
- The command:  
`BEGIN R := X; X := Y; Y := R END`  
 would fulfil the specification and so the contract
- The command:  
`BEGIN X := Y; Y := X END`  
 would not
- How do we determine whether a program fulfils the contract?

## Examples

$\{X = 1\} Y := X \{Y = 1\}$

- this says that if the command  $Y := X$  is executed in a state satisfying the condition  $X = 1$
- i.e. a state in which the value of X is 1
- *then*, if the execution terminates (which it does), the condition  $Y = 1$  will hold
- clearly this specification is true

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

- this says that if the execution of:  
 $Y := X$   
 terminates when started in a state satisfying  $X = 1$
- then  $Y = 2$  will hold
- this is clearly false

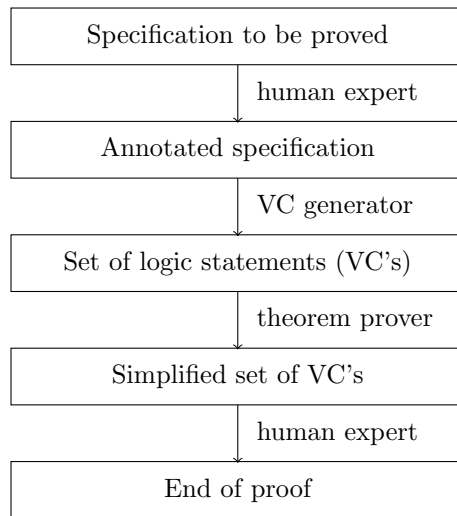
$\{X = 1\} \text{WHILE T DO SKIP } \{Y = 2\}$

- this specification is true!

## Hoare Logic and Verification Conditions

- Can prove  $\{P\} C \{Q\}$  by constructing a proof in Hoare Logic
  - original proposal by Hoare
  - tedious and error prone
  - impractical for large programs
- Can ‘compile’ proving  $\{P\} C \{Q\}$  to *verification conditions*
  - more natural
  - basis for computer assisted verification
- Proof of verification conditions equivalent to proof with Hoare Logic
  - Hoare logic can be used to explain verification conditions

### Architecture of a Verifier



### Verification Flow

- Input: a partial correctness specification annotated with mathematical statements
  - these annotations describe relationships between variables
- The system generates a set of purely mathematical statements called *verification conditions* (or VC's)
- If the verification conditions are provable, then the original specification can be deduced from the axioms and rules of Floyd-Hoare logic
- The verification conditions are passed to a *theorem prover* program which attempts to prove them automatically

## 2.4 Total Correctness

### Total Correctness

- 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
  - whenever  $C$  is executed in a state satisfying  $P$  the execution of  $C$  *terminates*
  - after  $C$  terminates  $Q$  holds

### Examples

$$[X = 1] Y := X [Y = 1]$$

- this says that *if* the command  $Y := X$  is executed in a state satisfying the condition  $X = 1$
- *then*, the execution *will* terminate
- and the condition  $Y = 1$  will hold of the final state

- clearly this specification is true

$[X = 1] Y := X; \text{ WHILE } T \text{ DO SKIP } [Y = 1]$

- this says that the execution of:  
 $Y := X; \text{ WHILE } T \text{ DO SKIP}$   
 terminates when started in a state satisfying  $X = 1$
- after which  $Y = 1$  will hold
- this is clearly false

### Total Correctness

- Informally: *total correctness* = *Termination* + *Partial correctness*
- Total correctness is the ultimate goal
  - usually easier to show partial correctness and termination separately
- Termination is usually straightforward to show, but there are examples where it is not: no one knows if the program below terminates for all values of  $X$ :  
 $\text{WHILE } X > 1 \text{ DO}$   
 $\quad \text{IF ODD}(X) \text{ THEN } X := (3 \times X) + 1$   
 $\quad \text{ELSE } X := X \text{ DIV } 2$
- the expression  $X \text{ DIV } 2$  evaluates to the result of rounding down  $X/2$  to a whole number
- Exercise: Write a specification which is true if and only if the program above terminates

## 2.5 Auxiliary Variables

### Auxiliary Variables

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

- 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, are used to name the initial values of program variables  $X$  and  $Y$
- They are called *auxiliary* variables or *ghost* variables
- Informal convention:
  - program variables are upper case
  - auxiliary variables are lower case

**Examples**

$$\{X = x \wedge Y = y\} \text{ BEGIN } X := Y; Y := X \text{ END } \{X = y \wedge Y = x\}$$

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

$$\{T\} C \{Q\}$$

- this says that whenever `C` halts, `Q` holds

$$\{P\} C \{T\}$$

- this specification is true for every condition `P` and every command `C`
- because `T` is always true

$$[P] C [T]$$

- this says that `C` terminates if initially `P` holds
- it says nothing about the final state

$$[T] C [P]$$

- this says that `C` always terminates and ends in a state where `P` holds

**A More Complicated Example**

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

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

This is:  $\{T\} C \{R < Y \wedge X = R + (Y \times Q)\}$

- where `C` is the command indicated by the brace above
- the specification is true if whenever the execution of `C` halts, then `Q` is the quotient and `R` is the remainder resulting from dividing `Y` into `X`
- it is true (even if `X` is initially negative!)
- in this example `Q` is a program variable

**Some Easy Exercises**

- When is  $[T] C [T]$  true?
- Write a partial correctness specification which is true if and only if the command `C` has the effect of multiplying the values of `X` and `Y` and storing the result in `X`
- Write a specification which is true if the execution of `C` always halts when execution is started in a state satisfying `P`

## 2.6 Tricky Specifications

### Specification can be Tricky

- “The program must set  $Y$  to the maximum of  $X$  and  $Y$ ”
  - $[T] C [Y = \max(X, Y)]$
- A suitable program:
  - IF  $X \geq Y$  THEN  $Y := X$  ELSE SKIP
- Another?
  - IF  $X \geq Y$  THEN  $X := Y$  ELSE SKIP
- Or even?
  - $Y := X$
- Later we will be able to prove that these programs are “correct”
- The postcondition “ $Y = \max(X, Y)$ ” says “ $Y$  is the maximum of  $X$  and  $Y$  *in the final state*”

### Specification can be Tricky

- The intended specification was probably not properly captured by:
  - $\{T\} C \{Y = \max(X, Y)\}$
- The correct formalisation of what was intended is probably:
  - $\{X = x \wedge Y = y\} C \{Y = \max(x, y)\}$
- The lesson:
  - it is easy to write the wrong specification!
  - a proof system will not help since the incorrect programs could have been proved “correct”
  - testing would have helped!

### Sorting

- Suppose  $C_{sort}$  is a command that is intended to sort the first  $n$  elements of an array
- To specify this formally, let  $SORTED(A, n)$  mean:
  - $A(1) \leq A(2) \leq \dots \leq A(n)$
- A first attempt to specify that  $C_{sort}$  sorts is:
  - $\{1 \leq N\} C_{sort} \{SORTED(A, N)\}$
- Not enough:
  - $SORTED(A, N)$  can be achieved by simply zeroing the first  $N$  elements of  $A$

### Permutation Required

- It is necessary to require that the sorted array is a rearrangement, or permutation, of the original array
- To formalise this, let  $\text{PERM}(A, A', N)$  mean that:

$$A(1), A(2), \dots, A(n)$$

is a rearrangement of:

$$A'(1), A'(2), \dots, A'(n)$$

- An improved specification that  $C_{\text{sort}}$  sorts:

$$\{1 \leq N \wedge A = a\}$$

$$\begin{array}{c} C_{\text{sort}} \\ \{\text{SORTED}(A, N) \wedge \text{PERM}(A, a, N)\} \end{array}$$

### Still Not Correct!

- The following specification is true:

$$\{1 \leq N\}$$

$$N := 1$$

$$\{\text{SORTED}(A, N) \wedge \text{PERM}(A, a, N)\}$$

- Must say explicitly that  $N$  is unchanged

- A better specification is thus:

$$\{1 \leq N \wedge A = a \wedge N = n\}$$

$$\begin{array}{c} C_{\text{sort}} \\ \{\text{SORTED}(A, N) \wedge \text{PERM}(A, a, N) \wedge N = n\} \end{array}$$

- Is this the correct specification?