Formal Specification

Overview

- **This lecture is concerned with** *formal specification*
- **Natural language specifications are often**
  - **inconsistent**
  - **ambiguous**
  - **difficult to follow (long rambling sentences)**
  - **cannot be reasoned about**

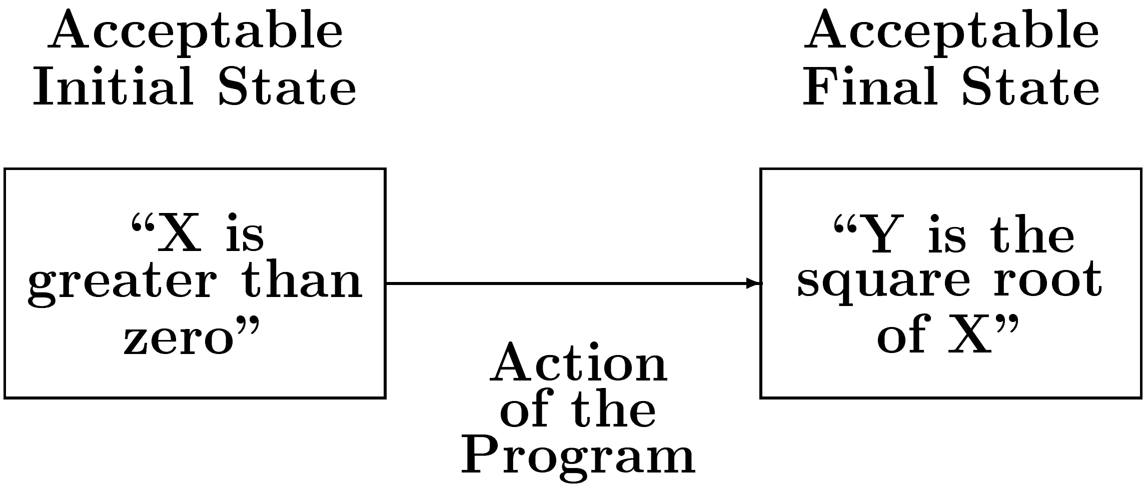
Formal specifications

- **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 Professor C.A.R. Hoare of Oxford**

Imperative Programs

- **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. 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**

Specification of Imperative Programs



Partial Correctness Specification

- **An expression $\{P\}\ C\ \{Q\}$ is called a** *partial correctness specification*
  - 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**

Some Examples

- $\{\mathtt{X} = 1\}$ `Y:=X` $\{\mathtt{Y} = 1\}$
  - **This says that *if* the command `Y:=X` is executed in a state satisfying the condition $\mathtt{X} = 1$**
  - **i.e. a state in which the value of `X` is 1**
  - ***then*, if the execution terminates (which it does)**
  - **then the condition $\mathtt{Y} = 1$ will hold**
  - **Clearly this specification is true**
- $\{\mathtt{X} = 1\}$ `Y:=X` $\{\mathtt{Y} = 2\}$
  - **This says that if the execution of `Y:=X` terminates when started in a state satisfying $\mathtt{X} = 1$**
  - **then $\mathtt{Y} = 2$ will hold**
  - **This is clearly false**
- $\{\mathtt{X} = 1\}$ `WHILE T DO SKIP` $\{\mathtt{Y} = 2\}$
  - **This specification is true!**

Total Correctness Specification

- **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**, then the execution of** C **terminates**
  - **After** C **terminates** Q **holds**

Some Examples

- $[\mathtt{X} = 1]$ `Y:=X` $[\mathtt{Y} = 1]$

  - **This says that *if* the command `Y:=X` is executed in a state satisfying the condition $\mathtt{X} = 1$**
  - ***then*, the execution *will* terminate**
  - **and the condition $\mathtt{Y} = 1$ will hold of the final state**
  - **Clearly this specification is true**

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

  - **This says that the execution of `Y:=X;WHILE T DO SKIP` terminates when started in a state satisfying $\mathtt{X} = 1$**
  - **after which $\mathtt{Y} = 1$ 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 whether the program below terminates for all values if `X`**

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

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

- **Exercise: Write a specification which is true if and only if the program above terminates**

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 and are used to name the initial values of program variables `X` and `Y`**

- **They are called *auxiliary* variables**

  - **also called *ghost* variables**

- **Informal convention:**

  - **Program variable are upper case**

  - **Auxiliary variable are lower case**

More simple examples

- $\{\texttt{X=x} \wedge \texttt{Y=y}\}$ `BEGIN X:=Y; Y:=X END` $\{\texttt{X=y} \wedge \texttt{Y=x}\}$
  - **This says that `BEGIN X:=Y; Y:=X END` exchanges the values of `X` and `Y`**
  - **This is not true**
- $\{\texttt{T}\}\ C\ \{Q\}$
  - **This says that whenever C halts, Q holds**
- $\{P\}\ C\ \{\texttt{T}\}$
  - **This specification is true for every condition P and every command C**
  - **Because `T` is always true**
- $[P]\ C\ [\texttt{T}]$
  - **This says that C terminates if initially P holds**
  - **It says nothing about the final state**
- $[\texttt{T}]\ C\ [P]$
  - **This says that C always terminates and ends in a state where P holds**

A More Complicated Example

- $\{\mathtt{T}\}$

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

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

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

  - **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 previous examples to range over postconditions**

Some Easy Exercises

- **When is** $[\mathtt{T}]\ C\ [\mathtt{T}]$ **true?**

- **Write a partial correctness specification which is true if and only if the command** C **has the effect of multiplying the values of** $\mathtt{X}$ **and** $\mathtt{Y}$ **and storing the result in** $\mathtt{X}$

- **Write a specification which is true if the execution of** C **always halts when execution is started in a state satisfying** P

Specification can be Tricky

- **"The program must set `Y` to the maximum of `X` and `Y`"**
  - $[\mathtt{T}]\ C\ [\mathtt{Y} = \mathtt{max(X,Y)}]$
- **A suitable program:**
  - `IF X >= Y THEN Y := X ELSE SKIP`
- **Another?**
  - `IF X >= Y THEN X := Y ELSE SKIP`
- **Or even?**
  - `Y := X`
- **Later you 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 (ii)

- **The intended specification was probably *not* properly captured by**

  $\vdash$ {T} C {Y=max(X,Y)}

- **The correct formalisation of what was intended is probably**

  $\vdash$ {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 $\mathtt{C}_{sort}$ is a command that is intended to sort the first n elements of an array**

- **To specify this formally, let $\mathtt{SORTED}(A, n)$ mean**

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

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

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

- **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** `PERM`(A, A', N) **mean that**

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

  is a rearrangement of

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

- **An improved specification that** $\mathtt{C}_{sort}$ **sorts:**

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

Still Not Correct ...

- **The following specification is true**

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

- **Must say explicitly that N is unchanged**

- **A better specification is thus:**

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

- **Is this the correct specification?**
  - **What if N is larger than the size of the array?**

Summary

- **We have given a notation for specifying**
  - **the partial correctness of programs**
  - **the total correctness of programs**
- **Specifications can still be incorrect**
  - **However, their meaning is precise**
- **We next describe the predicate calculus:**
  - **The precise language upon which they are based**

Review of Predicate Calculus

- **Program states are specified with:**
  - *first-order predicate calculus*
- **Knowledge of this is assumed**
  - **a brief review will be given**
- **Higher order predicate calculus will be:**
  - **used to** *semantically embed* **Floyd-Hoare logic**
  - **used for hardware specification**
  - **covered later**

Terms and Statements

- **In first order logic there are two separate syntactic classes**
  - **Terms (or expressions)**
    - **these denote values (e.g. numbers)**
  - **Statements (or formulae)**
    - **these are either true or false**
- **In higher order logic statements are a special case of terms**
  - **because truth-values are values**

Terms (Expressions)

- **Statements are built out of *terms* which denote *values* such as numbers, strings and arrays**

- **Terms, like `1` and $4 + 5$, denote a fixed value**

- **Other terms contain *variables* like `X`, `Y`, `Z` etc**

- **We will use conventional mathematical notation for terms, e.g.**

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

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

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

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

Atomic Statements

- **Examples of atomic statements are**

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

- `T` **and** `F` **are atomic statements that are always true and false**

- **Other atomic statements are built from terms using** *predicates*, **e.g.**

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

- `ODD` **and** `PRIME` **are examples of predicates**

- $=$ **and** $\geq$ **are examples of** *infixed* **predicates**

- `X`, `1`, `3`, `X+1`, $\texttt{(X+1)}^2$, $\texttt{X}^2$ **are examples of terms**

Compound statements

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

$$\begin{array}{ll} \neg & \text{(not)} \\ \wedge & \text{(and)} \\ \vee & \text{(or)} \\ \Rightarrow & \text{(implies)} \\ \Leftrightarrow & \text{(if and only if)} \end{array}$$

- **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**
  - $P \Leftrightarrow Q$ **is true if P and Q are either both true or both false**

More on Implication

- **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 $\neg P \vee Q$**

- **Some philosophers disagree with this treatment of implication**

  - **since any implication $A \Rightarrow B$ is true if A is false**

  - **e.g.** $(1 < 0) \ \Rightarrow \ (2 + 2 = 3)$

- $P \Leftrightarrow Q$ **is equivalent to** $(P \Rightarrow Q) \wedge (Q \Rightarrow P)$

Precedence

- **To reduce the need for brackets it is assumed that**
  - $\neg$ **is more binding than** $\wedge$ **and** $\vee$
  - $\wedge$ **and** $\vee$ **are more binding than** $\Rightarrow$ **and** $\Leftrightarrow$

- **For example**

$$\begin{array}{lll} \neg P \wedge Q & \text{is equivalent to} & (\neg P) \wedge Q \\ P \wedge Q \Rightarrow R & \text{is equivalent to} & (P \wedge Q) \Rightarrow R \\ P \wedge Q \Leftrightarrow \neg R \vee S & \text{is equivalent to} & (P \wedge Q) \Leftrightarrow ((\neg R) \vee S) \end{array}$$

Universal quantification

- **If S is a statement and x a variable**
- **Then $\forall x.\ S$ means:**

  'for all values of x, the statement S is true'

- **The statement**

$$\forall x_1\ x_2\ \ldots\ x_n.\ S$$

  abbreviates

$$\forall x_1.\ \forall x_2.\ \ldots\ \forall x_n.\ S$$

- **It is usual to adopt the convention that any unbound (i.e. *free*) variables in a statement are to be regarded as implicitly universally quantified**
- **For example, if n is a variable then the statement $n+0=n$ is regarded as meaning the same as $\forall n.\ n+0=n$**

Example

- **Let us abbreviate**

$$x_1 \leq x_2 \;\wedge\; x_2 \leq x_3 \;\wedge\; \ldots \;\wedge\; x_{n-1} \leq x_n$$

  by

$$x_1 \leq x_2 \leq x_3 \leq \; \ldots \; \leq x_{n-1} \leq x_n$$

- **Recall that** $\mathtt{SORTED}(A, N)$ **means that**

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

- **The definition of** `SORTED` **can now be given formally**

$$\begin{array}{l} \mathtt{SORTED}(A, N) \;\equiv \\ \quad \forall I\ J.\ 1 \leq I \leq J \leq N \;\Rightarrow\; A(I) \leq A(J) \end{array}$$

Comment on Definition of $\texttt{SORTED}$

- **The symbol $\equiv$ is synonymous with $\Leftrightarrow$ and means 'if and only if'**
  - **It is used to introduce the definitions of constants**

- **Notice that A and N are unbound variables in**

$$\begin{array}{l} \texttt{SORTED}(A,N) \ \equiv \\ \quad \forall I\ J.\ 1 \leq I \leq J \leq N \ \Rightarrow\ A(I) \leq A(J) \end{array}$$

- **Hence they are hence regarded as implicitly universally quantified**

- **The definition of $\texttt{SORTED}$ could thus have been written as**

$$\begin{array}{l} \forall A\ N.\ \texttt{SORTED}(A,N) \ \equiv \\ \qquad\qquad \forall I\ J.\ 1 \leq I \leq J \leq N \ \Rightarrow\ A(I) \leq A(J) \end{array}$$

Existential quantification

- **If S is a statement and x a variable**
- **Then $\exists x.\ S$ means**

  'for some value of x, the statement S is true'

- **The statement**

$$\exists x_1\ x_2\ \ldots\ x_n.\ S$$

  abbreviates

$$\exists x_1.\ \exists x_2.\ \ldots\ \exists x_n.\ S$$

Summary

- **Predicate calculus forms the basis for program specification**
- **It is used to describe the acceptable initial states, and intended final states of programs**
- **We will next look at how to prove programs meet their specifications**
- **We assume you know how to prove theorems within predicate calculus**