Formal Specification

Hoare Logic and Verification Conditions

- **Hoare Logic is a deductive proof system for Hoare triples $\{P\}\ C\ \{Q\}$**

- **Can use Hoare Logic directly to verify programs**
  - 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

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**

- $\{\mathtt{X} = 1\}$ `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 **the execution of** C **terminates**
  - **after** C **terminates** Q **holds**

- $[\texttt{X} = 1]$ `Y:=X; WHILE T DO SKIP` $[\texttt{Y} = 1]$
  - **this says that the execution of** `Y:=X;WHILE T DO SKIP` **terminates when started in a state satisfying** $\texttt{X} = 1$
  - **after which** $\texttt{Y} = 1$ **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 of** `X`

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

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

  - **the Collatz conjecture is that this terminates with** `X=1`

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 or *ghost* variables**
- **Informal convention:**
  - **program variable are upper case**
  - **auxiliary variable are lower case**

More simple examples

- $\{$`X=x` $\wedge$ `Y=y`$\}$ `BEGIN X:=Y; Y:=X 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**
- $\{\mathtt{T}\}\ C\ \{Q\}$
  - **this says that whenever C halts, Q holds**
- $\{P\}\ C\ \{\mathtt{T}\}$
  - **this specification is true for every condition P and every command C**
  - **because `T` is always true**
- $[P]\ C\ [\mathtt{T}]$
  - **this says that C terminates if initially P holds**
  - **it says nothing about the final state**
- $[\mathtt{T}]\ C\ [P]$
  - **this says that C always terminates and ends in a state where P holds**

A More Complicated Example

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

- **This is $\{\mathtt{T}\}\ \mathtt{C}\ \{\mathtt{R} < \mathtt{Y} \ \wedge\ \mathtt{X} = \mathtt{R} + (\mathtt{Y} \times \mathtt{Q})\}$**
  - **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 `Q` is a program variable**
  - **don't confuse `Q` with the metavariable Q used in previous examples to range over postconditions (Sorry: my bad notation!)**

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** `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

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

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

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

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

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

- **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** $\texttt{PERM}(A, A', N)$ **mean that**

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

  is a rearrangement of

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

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

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

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

- **Is this the correct specification?**

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 logic* (FOL)**
- **Knowledge of this is assumed (brief review given now)**
- **Later *higher-order logic* (HOL) will be:**
  - **used to *semantically embed* Floyd-Hoare logic**
  - **used for hardware specification**
- **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, and are called** *ground*

- **Other terms contain** *variables* **like** `x`, `X`, `y`, `X`, `z`, `Z` **etc**

- **We use conventional notation, e.g. here are some terms:**

$$
\begin{array}{c}
\mathtt{X}, \quad \mathtt{y}, \quad \mathtt{Z}, \\
\mathtt{1}, \quad \mathtt{2}, \quad \mathtt{325}, \\
\mathtt{-X}, \quad \mathtt{-(X+1)}, \quad \mathtt{(x \times y)+Z}, \\
\sqrt{\mathtt{(1+x^2)}}, \quad \mathtt{X!}, \quad \mathtt{sin(x)}, \quad \mathtt{rem(X,Y)}
\end{array}
$$

- **Convention:**
  - **program variables are uppercase**
  - **auxiliary (i.e. logical) variables are lowercase**

Atomic Statements

- **Examples of atomic statements are**

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

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

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

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

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

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

- `X`, `1`, `3`, `X+1`, $\texttt{(X+1)}^2$, $\texttt{x}^2$ **are terms in above atomic statements**

Compound statements

- **Compound statements are built up from atomic statements using:**

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

  - **The single arrow $\rightarrow$ is commonly used for implication instead of $\Rightarrow$**

- **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. $(1 < 0) \;\Rightarrow\; (2 + 2 = 3)$**
  - **search web for "paradoxes of implication"**

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

- **Sometimes write $P = Q$ or $P \equiv Q$ for $P \Leftrightarrow Q$**
  - **$P = Q$ used in higher-order logic**
  - **$P \equiv Q$ used for definitions**
  - **also $P =_{def} Q$ used for definitions**

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** $\texttt{SORTED}(A, N)$ **means that**

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

- **The definition of** $\texttt{SORTED}$ **can now be given formally**

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

Comment on Definition of $\mathtt{SORTED}$

- **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**

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

- **Unbound variables are regarded as implicitly universally quantified**

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

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

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

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

$PERM(A, a, N)$

- $PERM(A, a, N)$ **easy to define if one can quantify over functions**

$$
\begin{aligned}
& PERM(A,a,N) = \\
& \quad \exists f.\ \ (\forall n.\ n \leq N \Rightarrow f(n) \leq N)\ \wedge \\
& \qquad\quad (\forall m\ n.\ m \leq N \wedge n \leq N \wedge f(m) = f(n) \Rightarrow m = n)\ \wedge \\
& \qquad\quad (\forall n.\ n \leq N \Rightarrow a(n) = A(f(n)))
\end{aligned}
$$

- **Exercise: specify** $PERM(A, a, N)$ **in first-order logic**

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**

- **Proof of theorems within predicate calculus assumed known!**

Floyd-Hoare Logic

- **To construct formal proofs of partial correctness specifications,** *axioms* **and** *rules of inference are needed*

- **This is what Floyd-Hoare logic provides**
  - **the formulation of the deductive system is due to Hoare**
  - **some of the underlying ideas originated with Floyd**

- **A proof in Floyd-Hoare logic is a sequence of lines, each of which is either an** *axiom* **of the logic or follows from earlier lines by a** *rule of inference* **of the logic**
  - **proofs can also be trees, if you prefer**

- **A formal proof makes explicit what axioms and rules of inference are used to arrive at a conclusion**

Notation for Axioms and Rules

- **If S is a statement, $\vdash S$ means S has a proof**
  - **statements that have proofs are called *theorems***
- **The axioms of Floyd-Hoare logic are specified by *schemas***
  - **these can be *instantiated* to get particular partial correctness specifications**
- **The inference rules of Floyd-Hoare logic will be specified with a notation of the form**

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

  - **this means the *conclusion* $\vdash S$ may be deduced from the *hypotheses* $\vdash S_1$, $\ldots$, $\vdash S_n$**
  - **the hypotheses can either all be theorems of Floyd-Hoare logic**
  - **or a mixture of theorems of Floyd-Hoare logic and theorems of mathematics**

An example rule

The sequencing rule

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

- **If a proof has lines matching** $\vdash\ \{P\}\ C_1\ \{Q\}$ **and** $\vdash\ \{Q\}\ C_2\ \{R\}$
- **One may deduce a new line** $\vdash\ \{P\}\ C_1;C_2\ \{R\}$
- **For example if one has deduced:**

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

  $\vdash$ `{X=2} X:=X+1 {X=3}`
- **One may then deduce:**

  $\vdash$ `{X=1} X:=X+1; X:=X+1 {X=3}`
- **Method of verification conditions (VCs) generates** *proof obligation*

  $\vdash$ `X=1` $\Rightarrow$ `X+(X+1)=3`
  - **VCs are handed to a theorem prover**
  - **"Extended Static Checking" (ESC) is an industrial example**

Invalid ‘Proofs’

- **It is quite easy to come up with plausible rules for reasoning about programs that are actually unsound**
  - **some examples for `FOR`-commands are given later**
- **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**
- **Soundness and completeness will be discussed informally later**
- **Here is a bogus proof**

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 $\sqrt{1} = 1$.

- **See link** *Some classic fallacies* **on course web page**