Lecture 7:
Introduction to formal specifications

Lecture notes by Mike Gordon are used

Recall some definitions

- *Formal Specification* - **using mathematical notation to give a precise description of what a program should do**

- *Formal Verification* - **using precise rules to mathematically prove that a program satisfies a formal specification**

- *Formal Development (Refinement)* - **developing programs in a way that ensures mathematically they meet their formal specifications**

Introduction

- Verification of programs is based on formal specification and on related verification method.

  We will use Floyd-Hoare logic (FHL)

- Proof systems of the FHL style depend on particular programming language with its syntax and semantics

- In this course we will deal with the verification of

  - deterministic sequential *while*-programs;
  - non-deterministic sequential *while*-programs
  - parallel programs with shared variables;
  - parallel programs with message passing.

Programs as state transition systems

- Programs are structured specifications of state transition systems.
- Programming language defines constructs for specifying single transitions and transition compositions.
- State components are referred in conditions of command constructs like *if*-, *while*-, *for*-, *case*-command etc.

Some notations

- Programs are built out of *commands* like assignment, *if*-, *while*-, *for*-, *case*-command etc
- The terms '*program'* and '*command*' are synonymous.
- '*Program*' will only be used for commands representing complete algorithm.
- The '*statement*' is used for conditions on program variables that occur in correctness specifications.

Imperative programs - state

- 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 in our course may have states consisting of other things than the values of variables (e.g. I/O).

Imperative programs - execution

- To use an imperative program
  - first establish a state,
    i.e. set some variables to have values of interest
  - then execute the program,
    (to transform the initial state into a final one)
  - inspect the values of variables in the final state to get the result.

Simple *while*-language

	% Expressions
• E ::= N | V | $E1$+$E2$ | $E1$−$E2$ | $E1$×$E2$ | …	% Arithmetic
• B ::= T | F | $E1$=$E2$ | $E1$≤$E2$ | …	% Logic
• C ::=	%Commands:
SKIP	% empty command (place holder)
| V := E	% assignment
| V(E1) := E2	% array assignment
| $C1$; $C2$	% sequential execution
| IF B THEN $C1$ ELSE $C2$	% conditional execution
| BEGIN VAR $V1$;…VAR Vn; C END	% block command (var. scoping)
| WHILE B DO C	% *while* - loop
| FOR V := $E1$ UNTIL $E2$ DO C	% *for* - loop

Terminology and notations

- *Variable*
  - `V1, V2, ..., Vn`
- *Program state* - valuation of program (and control) variables
- *Command* - gives a rule how the program state changes
  - `C1, C2, ... , Cn`
- *Program* - command that includes all the commands in the algorithm
  - `C`
- *Expression*
  - Arithmetic expression gives a value: `E1, E2, ... , En`
  - Boolean expression gives a *truth*-value: `B1, B2, ... , Bn`
- *Statement* – logical expression on program variables in the pre- and postconditions of the specification
  - `S1, S2, ... , Sn`

Formal specification

- Describes the intended behaviour of the program
- Specifies what the program <u>must do</u>
- Has well-defined *synax* and *semantics*
- that helps avoiding *ambiguous* and *controversial* specifications
- Can be used to prove the *correctness of the program*
- Can be used to generate *tests* and *counterexamples*

We will use formalism that is based on <u>FHL and predicate calculus</u>

Hoare’s notation

- **C.A.R. Hoare introduced the following notation called a** *partial correctness specification* **for specifying what a program does:**

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

where:

  - C **is a program from the programming language whose programs are being specified**
  - P **and** Q **are conditions on the program variables used in** C

Hoare's notation

- **Conditions on program variables will be written using standard mathematical notations together with *logical operators* like:**
  - $\wedge$ (**'and'**), $\vee$ (**'or'**), $\neg$ (**'not'**), $\Rightarrow$ (**'implies'**)

- **Hoare's original notation was $P\ \{C\}\ Q$ not $\{P\}\ C\ \{Q\}$, but the latter form is now more widely used**

Partial Correctness

- **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**

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)**
  - **then the condition $Y = 1$ will hold**
  - **Clearly this specification is true**

Examples

- $\{X = 1\}$ `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**
- $\{X = 1\}$ `WHILE T DO SKIP` $\{Y = 2\}$
  - **This specification is true!**

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

Example

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

$$\begin{aligned}&Total\ correctness =\\&\quad Termination + Partial\ correctness\end{aligned}$$

- **Total correctness is the ultimate goal**
  - **usually easier to show partial correctness and termination separately**

Total correctness

- 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
```

  - 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 in the specification

- $\{X=x \wedge Y=y\}$ `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**

Examples

- $\{\mathtt{X=x} \wedge \mathtt{Y=y}\}$ `BEGIN X:=Y; Y:=X END` $\{\mathtt{X=y} \wedge \mathtt{Y=x}\}$
  - **This says that `BEGIN X:=Y; Y:=X END` exchanges the values of `X` and `Y`**
  - **This is not true**

Examples

- $\{\mathtt{T}\}\ C\ \{Q\}$
  - **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 $\mathtt{T}$ is always true**

Examples

- $[P]\ C\ [\mathtt{T}]$
  - 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

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

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

- **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 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 (1)

- **"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`

Specification can be Tricky (2)

$$[\mathtt{T}]\ C\ [\mathtt{Y} = \mathtt{max(X,Y)}]$$

- **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"**

Specification can be Tricky (3)

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

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

- The correct formalisation of what was intended is probably

$$\vdash \{\mathtt{X{=}x} \wedge \mathtt{Y{=}y}\}\ \mathtt{C}\ \{\mathtt{Y{=}max(x,y)}\}$$

Specification can be Tricky (4)

- 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!

More Tricky example: 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**

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

Sorting: naive spec

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

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

- **Not enough:**
  - $\mathtt{SORTED(A,N)}$ **can be achieved by simply zeroing the first $\mathtt{N}$ elements of $\mathtt{A}$**

Sorting: permutation required

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

- **To formalise this, let** $\mathtt{PERM}(A, A', N)$ **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:**

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

Sorting: still not correct

- **The following specification is true**

$$\{1 \le \mathtt{N}\}$$
```
 N:=1
```
$$\{\mathtt{SORTED(A,N)} \land \mathtt{PERM(A,a,N)}\}$$

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

Sorting: still not correct

- **A better specification is thus:**

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

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

Summary

- **We have given a notation for specifying**
  - partial correctness of programs
  - total correctness of programs
- **It is easy to write incorrect specifications**
  - and we can prove the correctness of the incorrect programs
- **It is recommended to use testing, simulation and formal verification hand in hand.**