

# CS 3 2009 Lecture 3

Last time: propositional and predicate calculus.

Today:

- Applying logic to computer programs.

# Summary of Logic

- Propositional calculus.
- Atomic boolean expressions (“atoms”).
- “Strong” and “weak” statements.
- Quantifiers.

# Today

- Putting the logic to work.
- Question: what is programming?

Answer: the act of translating a declarative specification into an operational implementation.

Declarative specification:

- Compute the product of  $a[0] \dots a[5]$
- Compute the integral of the Schrödinger equation for the case. . .
- Compute whether or not the bridge can take the load of fifteen trucks.
- Given the lists of names A and B, return **true** if any name appears on both of them; else return **false**.

## Example

- “Write a program that multiplies together the elements of the array a.” (You happen to know they are all positive.)
- Your answer:

```
VAR
    a := ARRAY OF LONGREAL
        { 1.0d0, 3.0d0, 4.0d0 };
    ...
VAR
    prod := 1.0d0;
BEGIN
    FOR i := FIRST(a) TO LAST(a) DO
        prod := prod * a[i]
    END
END
```

## Alternative

- What if you turned in the following:

```
VAR
    a := ARRAY OF LONGREAL
        { 1.0d0, 3.0d0, 4.0d0 };
    ...
VAR
    sum := 0.0d0;
    prod : LONGREAL;
BEGIN
    FOR i := FIRST(a) TO LAST(a) DO
        sum := sum + Math.log(a[i])
    END;
    prod := Math.exp(sum)
END
```

- The program doesn't satisfy the specification: it doesn't "multiply together the elements of the array a."
- Is anyone likely to care?

# Conclusion

- The person that ordered the program asked for the wrong thing.
- He should have said, “Compute the product of the elements of a.”
- If he’d been paying attention in CS 3, he would have said, “write a program that establishes the truth of the expression

$$\text{prod} = \prod_{i=f}^l a[i]$$

where  $f = \text{FIRST}(a)$  and  $l = \text{LAST}(a)$ .”

- This kind of statement is called a *postcondition*.

## But wait...

There's a problem. We said that "you happen to know they are all positive."

So,

$$\text{prod} = \prod_{i=f}^l a[i]$$

is not enough of a specification, because it doesn't rule out having an  $a[i]$  non-positive.

- For a complete specification, we have to include the *precondition*

$$\forall i : f \leq i \leq l : a[i] > 0$$

(In this case, we could leave out  $f \leq i \leq l$ , since it's obvious.)

# Complete specification

“Write me a program  $S$  that satisfies:

If  $S$  is activated in a state where

$$\forall i :: a[i] > 0,$$

then  $S$  will terminate in a state where

$$\text{prod} = \prod_{i=f}^l a[i].”$$

More succinctly,

$$\{\forall i :: a[i] \geq 0\} \quad S \quad \{\text{prod} = \prod_{i=f}^l a[i]\}$$

- $\{P\}S\{Q\}$  is called a *Hoare triple*.

# Hoare triples

$$\{P\}S\{Q\}$$

“If  $S$  is activated in a state where  $P$  holds, then the execution of  $S$  will terminate in a state where  $Q$  holds.”

*Example.*

$$\{x = 0\}x := 1\{x = 1\}$$

How can this go wrong?

- $S$  might terminate in a state where  $Q$  doesn't hold.

$$\{x = 0\}x := 1\{x = 2\}$$

(false)

- $S$  might not terminate: it might go into an infinite loop, or it might contain an error of some sort, which would cause the execution to be *aborted*.

$$\{x = 0\}\text{WHILE } x \neq 1 \text{ DO } x := x - 1 \text{ END}\{x = -2\}$$

(false)

$$\{x = 0\}x := x/x\{x = 1\}$$

(false)

# Ghost variables

The statements and variables that appear in the pre- and postconditions need not appear explicitly in the program text.

- Consider a sorting program:

$$\{a \text{ is an array of integers}\} \textit{sort} \{a \text{ is sorted}\}$$

- You might want to talk about, for instance,

$$m = \sum_{i=f}^l i a[i]$$

( $m$  must be minimized if the array is sorted.)

- $m$  is cumbersome to compute; there is no reason to, either.
- A variable introduced for the purpose of describing a program but not appearing in the program text is called a *ghost variable*.

# Hoare triples

$$\{P\}S\{Q\}$$

- $P$  is called the precondition or *input assertion*.
- $Q$  is called the postcondition or *output assertion*.
- The triple says nothing about what will happen if  $S$  is activated in a state where  $P$  doesn't hold.

In practice,

- Pre- and postconditions are usually stated in the program text as comments.
- One of the most useful language constructs is the ability to leave assertions in the code. In Modula-3:

```
<* ASSERT x > 0 *>  
x := x + 1;  
<* ASSERT x > 1 *>
```

If the assertions do not hold at runtime, the program will be aborted. (Less useful for ghost variables.)

# Documentation

The Hoare triple is very popular in “software engineering”:

- Some call the pre- and postconditions *requires/ensures* or *requires/effects* clauses.
- Elaborate documentation systems exist that allow programmers to document their code in terms of pre- and postconditions (e.g., javadoc)
- Some integrate the documentation with the program execution (Eiffel). (Again, less useful for ghost variables.)

## What's this for?

- Hoare triples can be (and are) used at every level when describing imperative or object-oriented programs. Large programs:

“Design a program that, if activated in a state where the array `t` contains the cross-section of the trusses used in the bridge, will terminate in a state where `ok` contains `true` if the bridge meets the strength requirements and `false` otherwise.”

- Small programs (the following are universally true):

$$\{R\}\text{skip}\{R\}$$

(If the program `skip` is activated in a state satisfying  $R$ , then it will terminate in a state satisfying  $R$ ; for all  $R$ .)

$$\{\text{false}\}\text{abort}\{R\}$$
$$\{R\}x := a\{R_{a \rightarrow x}\}$$

# The if-statement

One last small program:

$$\{P_1\}S_1\{Q_1\} \wedge \{P_2\}S_2\{Q_2\} \Rightarrow$$
$$\{(P_1 \wedge B) \vee (P_2 \wedge \neg B)\}$$

IF  $B$  THEN  $S_1$  ELSE  $S_2$  END

$$\{Q_1 \vee Q_2\}$$

Usually useful when  $Q_1$  and  $Q_2$  are similar.

$$\{x < y\}swap\{x > y\} \wedge \{x \geq y\}skip\{x \geq y\} \Rightarrow$$
$$\{\mathbf{true}\}$$

IF  $x < y$  THEN  $swap$  ELSE  $skip$  END

$$\{x \geq y\}$$

- Can entirely formalize a programming language.
- We won't do that in CS 3 .

## Making useful assertions

Usually, the values dealt with by the program are not constants known *a priori*. In those cases, we have to introduce names for the unknown values.

*Example.* Specifying the program `swap(x, y)`

$$\{x = A \wedge y = B\} \text{swap} \{x = B \wedge y = A\}$$

Equivalent to

$$\forall A, B :: (\{x = A \wedge y = B\} \text{swap} \{x = B \wedge y = A\})$$

(See how important the rule  $(F \Rightarrow F) = T$  is?)

# What programming is all about

$$\{P\}S\{Q\}$$

Your task as programmer is to design the mechanism that takes us from  $P$  to  $Q$ .

You almost always have to decompose the problem, introducing intermediate states, say  $R$ :

$$\{P\}S_1\{R\}S_2\{Q\}$$

*Example.* Sort  $x, y, z$  so that  $x \leq y \leq z$ .

$$\{x = X \wedge y = Y \wedge z = Z\}$$

$S$

$$\{x \leq y \leq z \wedge x, y, z = \text{perm}(X, Y, Z)\}$$

*Solution.* Engage in some wishful thinking. Introduce

$$R = (z = \max(X, Y, Z) \wedge x, y, z = \text{perm}(X, Y, Z));$$

then we have

$$\{x = X \wedge y = Y \wedge z = Z\}$$

$S_1$

$$\{z = \max(X, Y, Z) \wedge x, y, z = \text{perm}(X, Y, Z)\}$$

$S_2$

$$\{x \leq y \leq z \wedge x, y, z = \text{perm}(X, Y, Z)\}$$

- *Note carefully.* It's almost always easier to engage in wishful thinking back from the solution than to try to go "forwards." (Same as recursive Scheme programs.)

Therefore,

$$\{R_{x \rightarrow a}\} x := a \{R\}$$

is probably more useful than

$$\{R\} x := a \{R_{a \rightarrow x}\}.$$

## Data types as assertions

Most programming languages require data types to be “declared”; for instance,

```
TYPE Array = ARRAY [0..9] OF INTEGER;
```

```
PROCEDURE S(a : Array;  
            dir : INTEGER) : Array =  
    (* Sort a in ascending order and return  
       the result if dir = 0;  
       if dir = 1, do the same same  
       but in descending order *)  
BEGIN  
    ...  
END Sort;
```

Array and INTEGER can be seen as part of the pre- and postconditions.

$$\{a \text{ is an Array } \dots\} r := S(a, d) \{r \text{ is an Array } \dots\}$$

- What happens if S is called with `dir= 2`?

# Data types

- In a language like Modula-3, careful use of data types can make your program clearer without adding ASSERT statements:

```
PROCEDURE S(a : Array;  
           dir : [0..1]) : Array =  
  (* Sort a in ascending order and return  
   the result if dir = 0;  
   if dir = 1, do the same same  
   but in descending order *)  
BEGIN  
  ...  
END Sort;
```

The declaration `dir : [0..1]` is equivalent to an assertion every time `dir` is used.

- A program can never be too clear!

# Summary

- Hoare triples: preconditions and postconditions; used for specification and program proofs.
- Simple examples: skip, abort, assignment, if.
- Start from where you want to go, and work backwards to where you are!
- Next time: loops.