

CS 3 2009 Lecture 4

Last time: specifications, implementations, Hoare triples.

Today:

- Loops.

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!

Loops

Some loops are easy to understand...

```
FOR i := 0 TO 2 DO
    prod := prod * a[i]
END
```

There's nothing new here: this loop is equivalent to

```
prod := prod * a[0];
prod := prod * a[1];
prod := prod * a[2]
```

In general, for-loops aren't difficult to deal with (at least not in Modula-3): it's usually easy to tell how many times they will be executed.

Often, each iteration is independent of the others; it wouldn't matter if we wrote

```
FOR i := 0 TO 2 DO
  c[i] := a[i] + b[i]
END
```

or

```
FOR i := 2 TO 0 BY -1 DO
  c[i] := a[i] + b[i]
END
```

- We won't worry too much about these kinds of loops.

While loops

The difficult loops are the while loops (or do-loops).

Example. Finding the index of an element (which is known to exist) in an array. (“Linear search.”)

```
(* e is the value we're looking for *)  
VAR  
  i := 0;  
BEGIN  
  WHILE a[i] # e DO  
    i := i + 1  
  END  
  (* i contains the index of an a[i]  
    s.t. a[i] = e *)  
END
```

Can we put this kind of program on as sure a footing as the other programs we've seen?

(Of course, this program is “obviously correct,” but we could imagine loops that are difficult to understand.)

The Loop Invariant

```
i := 0;
WHILE a[i] # e DO
    i := i + 1
END
```

We handle loops by introducing a predicate called the *loop invariant* (often just the *invariant* for short).

- The loop invariant is **true** at the beginning and end of each iteration. (By implication, it is **true** when the loop is first entered and **true** when the loop is finally exited.)
- *Example.* Introduce if to denote the index of the first $a[i]$ s.t. $a[i] = e$. (Of course, we don't know if before we run the program, but that's immaterial.)
- One loop invariant that might work here is $I = (i \leq if)$.

```
i := 0;
(* i <= if *)
WHILE a[i] # e DO
  (* i <= if *)
  i := i + 1
  (* i <= if *)
END
```

How do we prove that the program maintains the invariant?

Here's the argument:

- I is **true** initially since $if \geq 0$.
- The only way I could be made **false** is if we execute the statement $i := i + 1$ in a state where already $i = if$.
- This never happens because if $i = if$, then $a[i] = e$, by definition of if , in which case the loop exits.

So, we have shown:

```
i := 0;
(* i <= if *)
WHILE a[i] # e DO
  (* i <= if *)
  i := i + 1
  (* i <= if *)
END
```

Is that enough? No, we also need to show that the loop doesn't exit in any other state; that is, that it exits *only* when $i = if$.

- Assume that the loop exits for some i s.t. $i = ig < if$. That would mean that $a[ig] = e$, which contradicts the definition of if .
- Therefore, the loop exits only when $i = if$, the index of the first element $a[i]$ s.t. $a[i] = e$. *Q.E.D.*

Proof by Loop Invariant

General idea:

$$\{I\}\text{WHILE } G \text{ DO } \{I\}S\{I\} \text{ END}\{I \wedge \neg G\}$$

- I holds for every iteration (including the first and last);
 $\neg G$ holds only at the end of execution.

To show the correctness of the loop, you need to show that

- $I \wedge \neg G$ implies the desired postcondition and that
- I is implied by the desired precondition.

Note. “Invariant” doesn’t mean “always true”!

It means “true at beginning and end of each iteration”

In

$$\{I\}\text{WHILE } G \text{ DO } \{I\}S_1; S_2\{I\} \text{ END}\{I \wedge \neg G\},$$

it is not necessary (and often not possible) to demand that I hold between S_1 and S_2 .

Using Invariants

We've seen loop invariants as a way to prove the correctness of existing programs (that's how they are usually presented).

- Seems useless for “the working programmer”: the whole point is coming up with the program!
- Instead, think of the invariant as a way to get from the postcondition to the precondition.
- *Example.* Given the specification

$$\{\mathbf{true}\}L\{i = if\},$$

come up with L !

- Idea: first weaken the postcondition to come up with a suitable I , then write the loop

$$\{I\}\mathbf{WHILE} G \mathbf{DO} \{I\}S\{I\} \mathbf{END}\{I \wedge \neg G\}$$

that maintains I and exits when G is false.

Weakening the Postcondition

Ways of weakening the postcondition:

- Delete a conjunct.
- Replace a constant by a variable.
- Enlarge the range of a variable.
- Add a disjunct. (Useless)

For our search program, we weakened by enlarging the range of i from $i = if$ to $i \leq if$ and made the exit loop exactly when $i = if$.

GCD program

```
(* x = X AND y = Y *)  
WHILE x # y DO  
  IF x > y THEN  
    x := x - y  
  ELSE  
    y := y - x  
  END  
END;  
END;
```

Given the precondition $x = X \wedge y = Y$, what can we do with this program?

Lemma 1. $\text{GCD}(a, b) = \text{GCD}(a - b, b)$

Lemma 2. $\text{GCD}(a, a) = a$

GCD program

```
(* x = X AND y = Y *)  
WHILE x # y DO  
  IF x > y THEN  
    x := x - y  
  ELSE  
    y := y - x  
  END  
END  
END;
```

- Invariant: $\text{GCD}(x, y) = \text{GCD}(X, Y)$

There's a problem!

We've "proved" the correctness of an incorrect program!

```
(* x = X AND y = Y *)
WHILE x # y DO
  IF x > y THEN
    x := x - y
  ELSE
    y := y - x
  END
END
END;
```

We have proved: *if* the loop terminates, then $x = y = \text{GCD}(X, Y)$.

- That's not enough to prove the desideratum, namely that

$$\{x = X \wedge y = Y\}S\{x = y = \text{GCD}(X, Y)\}$$

because it doesn't show that S terminates.

Variant Function

We have to introduce a “variant function” or “bound function.”

Traditionally, the variant function has the following properties:

- It is an integer function.
- It is bounded below by zero.
- Every iteration of the loop reduces the bound function by at least one.

Variant Function for GCD

Let's introduce

$$V = x + y.$$

Can we prove that V has the desired properties?

- It is an integer function.
- It is *not* bounded below by zero.
- Every iteration of the loop does *not* reduce the bound function.

What do we do now?

- Strengthen the precondition: demand that

$$X > 0 \wedge Y > 0.$$

Now, V has the required properties.

We have proved

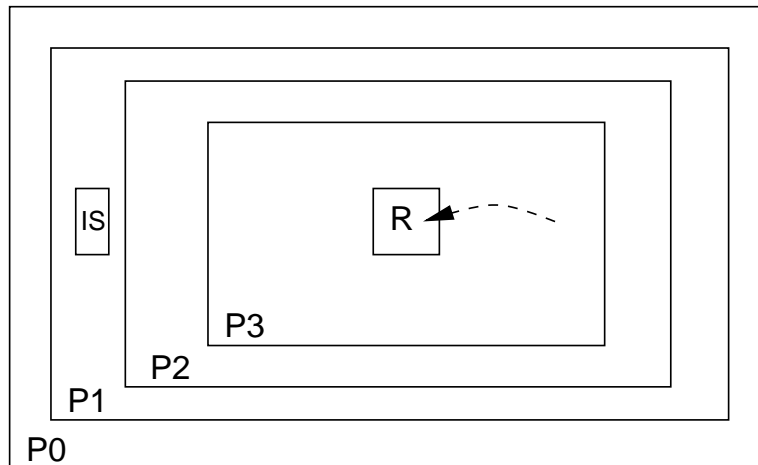
$$\{x = X \wedge y = Y \wedge Y > 0 \wedge X > 0\}$$

WHILE G DO S END

$$\{x = y = GCD(X, Y)\}$$

General idea

- “Balloon theory” (D. Gries):



- Start in initial state IS
- variant function shows you proceed from $P0$ through Pn
- eventually you must reach R .