

# Structured Programming

Valuable for what it prevents: arbitrary jumps and "spaghetti code".

Benefits:

- control constructs are few and simple
- sequencing is indicated by syntactic structure

Determination of which parts of a program affect a variable's value is simplified, and **formal reasoning** about programs is easier.

We take a brief look at how formal reasoning plays out in an imperative framework, in which referential transparency does not hold.

Instead of reasoning about the program text directly, we reason about the program's *state* at various points in the text.

## Formal Specifications

**Predicates:** Pre- and postconditions

$\{Q\} S \{R\}$  a Hoare triple;  $Q$  and  $R$  are predicates on the state;  $S$  is a statement

means

Provided  $Q$  (the *precondition*) holds initially, execution of  $S$  establishes  $R$  (the *postcondition*).

Example:

$\{y < 3\} x := y+1 \{x < 4\}$

Note: this formula is a **predicate** (true/false).

Proving its truth is an example of **program verification**.

## Weakest-precondition specifications

Reasoning about programs is **goal-directed**: We work **backwards** from a desired **postcondition**.

The foundation of one well-known technique is Dijkstra's predicate-transformation function **wp**:

$wp :: Statement \rightarrow Predicate \rightarrow Predicate$

$wp S R$  gives the **weakest precondition** for  $S$  such that  $S$  establishes  $R$ .

**wp** answers questions like

$\{ ? \} x := y+1 \{ x < 4 \}$

Why do we want "*weakest*" preconditions?

For any given  $S$  and  $R$ , there are usually many preconditions  $Q$  under which  $S$  establishes  $R$ .

Examples:

$\{ y = 2 \} x := y+1 \{ x < 4 \}$

$\{ y < 0 \} x := y+1 \{ x < 4 \}$

$\{ y = -77 \} x := y+1 \{ x < 4 \}$

...but most of them are more restrictive ("*stronger*") than they need to be.

The *least restrictive* ("*weakest*")  $Q$  is usually the most useful (and the most easily satisfied).

$\{ y < 3 \} x := y+1 \{ x < 4 \}$

## Dijkstra's Guarded-Command notation

An idealized imperative programming notation ("GCN") designed expressly to support formal reasoning.

The semantics of each type of GCN **command** (i.e., statement) is defined in terms of its effects...

not on program state (as is most common) but on **predicates** about the state.

The goal is to support formal *derivation* of programs and their correctness proofs from the programs' specifications.

First, a notation for specifying textual substitutions:

$(E)_{u,v,w}^{x,y,z}$  means

expression  $E$ ...

with each **free** occurrence of  $x$ ,  $y$ , and  $z$  replaced —*simultaneously*— by  $(u)$ ,  $(v)$ , and  $(w)$

Examples:

$$\left( \frac{-b + \sqrt{b^2 - 4ac}}{2a} \right) a, \quad b, \quad c \\ c+2, \quad a-1, \quad 3/b \\ = \frac{-(a-1) + \sqrt{(a-1)^2 - 4 \times (c+2) \times (3/b)}}{2 \times (c+2)}$$

$$\left( \left( \left( \frac{-b + \sqrt{b^2 - 4ac}}{2a} \right) a \right) c+2 \right) b \\ a-1 \quad 3/b \\ = \frac{-(a-1) + \sqrt{(a-1)^2 - 4 \times ((3/b)+2) \times (3/b)}}{2 \times ((3/b)+2)}$$

## GCN's Atomic commands

### 1. assignment ( $:=$ )

definition by example:

$$(wp \text{ "x,y,z := E}_0, E_1, E_2" R) = (R)_{E_0, E_1, E_2}^{x, y, z}$$

examples:

$$(wp \text{ "x := 3" } x \geq 3) = 3 \geq 3 = \text{true}$$

$$(wp \text{ "x := 2" } x \geq 3) = 2 \geq 3 = \text{false}$$

$$(wp \text{ "x := y+1" } x < 4)$$

$$= (x < 4)_{y+1}^x$$

$$= y+1 < 4$$

$$= y < 3$$

$$(wp \text{ "x,y := y+1,2" } x < y)$$

$$= (x < y)_{y+1, 2}^{x, y}$$

$$= y+1 < 2$$

$$= y < 1$$

### 2. skip: the *do-nothing* command

definition:  $wp \text{ skip } R = R$

### 3. abort: *error halt*

definition:  $wp \text{ abort } R = \text{false}$

GCN's **Composite commands****4. sequential composition (;)**

definition:

$$\text{wp } "S_0 ; S_1" R = \text{wp } S_0 (\text{wp } S_1 R)$$

The **Guarded Command**: $B \rightarrow S$  means $S$  (a command) is eligible for execution only if  $B$  (the *guard*) holds**5. selection:****if**  $B_0 \rightarrow S_0 \parallel B_1 \rightarrow S_1 \parallel \dots \parallel B_n \rightarrow S_n$  **fi**

- if all guards are false: = **abort**
- if two or more guards are true: one is selected (unpredictably)
- Note the absence of **else**.
- $(\text{wp } \text{"if } B_0 \rightarrow S_0 \parallel B_1 \rightarrow S_1 \parallel \dots \parallel B_n \rightarrow S_n \text{ fi"} R) = (\exists i \mid 0 \leq i \leq n : B_i) \wedge (\forall i \mid 0 \leq i \leq n : (B_i \Rightarrow \text{wp } S_i R))$

Example:

{ **true** }**if**  $x \leq y \rightarrow m := x \parallel x \geq y \rightarrow m := y$  **fi**{  $m = x \text{ min } y$  }**6. repetition:****do**  $B_0 \rightarrow S_0 \parallel B_1 \rightarrow S_1 \parallel \dots \parallel B_n \rightarrow S_n$  **od**

- iteration repeats while any guard is true
- all guards false: = **skip**
- two or more guards true: one is selected (unpredictably)
- **do**  $B_0 \rightarrow S_0 \parallel B_1 \rightarrow S_1 \parallel \dots \parallel B_n \rightarrow S_n$  **od**  
 $\{ \neg B_0 \wedge \neg B_1 \wedge \dots \wedge \neg B_n \}$

Example: Sorting four variables

{ **true** }**do**  $w > x \rightarrow w, x := x, w$  $\parallel x > y \rightarrow x, y := y, x$  $\parallel y > z \rightarrow y, z := z, y$ **od**{  $w \leq x \leq y \leq z$  }The **do...od** construct is actually defined by its wp function (not shown here, because it's not useful in code development or verification).

Example: Multiplication by repeated addition

{  $0 \leq n$  } $i, p := 0, 0;$ {  $0 \leq i \leq n \wedge p = i \cdot x$  }**do**  $i \neq n \rightarrow i, p := i+1, p+x$  **od**{  $p = n \cdot x$  }**Invariants: A Linear-search example**

(Sethi, §§ 3.5, 3.6)

Given:

- an array  $A[0 : n]$  where  $0 \leq n$
- a value  $x$  such that  $x \in A$

Find:

- the largest  $i$  such that  $0 \leq i \leq n \wedge A[i] = x$ .

The problem clearly calls for a loop.

The **general form** of a loop:{ $Q$ } initialize; **do**  $B \rightarrow S$  **od** { $R$ } $Q$  and  $R$  are given; our objective is to synthesize

- *initialize*
- the guard  $B$
- the loop body  $S$ .

A powerful technique: **loop invariants**. $P$  is *invariant* with respect to a command  $S$  iff{ $P$ }  $S$  { $P$ } $P$  is invariant with respect to a loop **do...od** iff $P$  holds *before* any iteration of the loop $\Rightarrow$  $P$  holds *after* that iterationFurthermore, when a loop terminates, its guard is **false**:**do**  $B \rightarrow S$  **od** { $\neg B$ }

In terms of the general form:

{ $Q$ }establish  $P$ ;{ $P$ }**do**  $B \rightarrow \{P \wedge B\} S \{P\}$  **od**{ $P \wedge \neg B$ } { $R$ }

Notes:

0.  $P \wedge \neg B \Rightarrow R$  helps in developing  $P$  and  $B$ .
1. Then we have pre- and postconditions for  $S$ .

Now the postcondition

$$R = (A[i] = x) \wedge (\forall j \mid i < j \leq n : A[j] \neq x)$$

breaks very nicely into

$$P = (\forall j \mid i < j \leq n : A[j] \neq x)$$

and

$$\neg B = (A[i] = x), \quad \text{i.e., } B = (A[i] \neq x)$$

(Note: in this case  $P \wedge \neg B = R$ .)

So now we have

{ $Q$ }establish  $P$ ;{  $P : (\forall j \mid i < j \leq n : A[j] \neq x)$  }**do**  $A[i] \neq x \rightarrow \{P \wedge A[i] \neq x\} S \{P\}$  **od**{ $P \wedge A[i] = x$ }{ $R$ }