

Logic Specifications Axiomatic Semantics

Outline

- Operational specifications
 - Data Flow Diagrams, Finite State Machines, Petri nets
- Descriptive specifications
 - Logic/algebraic specifications
 - Entity-Relationship Diagrams

Logic specifications

Examples of first-order theory (FOT) formulas:

- $x > y$ and $y > z$ implies $x > z$
- for all x, y, z ($x > y$ and $y > z$ implies $x > z$)
- $x + 1 < x - 1$
- for all x (exists y ($y = x + z$))
- $x > 3$ or $x < -6$

Specifying complete programs

A *property, or requirement*, for P is specified as a formula of the type

$$\{\text{Pre } (i_1, i_2, \dots, i_n) \}$$
$$P$$
$$\{\text{Post } (o_1, o_2, \dots, o_m, i_1, i_2, \dots, i_n) \}$$

Pre: precondition

Post: postcondition

Example

- Program to compute greatest common divisor

$\{i1 > 0 \text{ and } i2 > 0\}$

P

$\{(exists\ z1, z2\ (i1 = o * z1 \text{ and } i2 = o * z2)$

and not (exists h

(exists z1, z2 (i1 = h * z1 and i2 = h * z2) and h > o))\}

Specifying procedures

```
{n > 0} -- n is a constant value  
procedure search (table: in integer_array;  
                  n: in integer;  
                  element: in integer;  
                  found: out Boolean);  
{found = (exists i (1 ≤ i ≤ n and table (i) = element))}
```

```
{n > 0 }  
procedure reverse (a: in integer_array;  
                  n: in integer;  
                  b: out integer_array);  
{for all i (1 ≤ i ≤ n) implies (b (i) = a (n - i + 1))}
```

Exercises

```
{ ? }  
procedure sort (a: in out integer_array;  
                n: in integer);  
{ ? }
```

```
{ ? }  
procedure substring (a: in char_array;  
                    start: in integer;  
                    length: in integer  
                    b: out char_array);  
{ ? }
```

Exercises

```
{ ? }  
procedure strcat (a: in char_array;  
                  alen: in integer;  
                  b: in char_array;  
                  blen: in integer;  
                  c: out char_array;  
                  clen: in integer;  
  
{ ? }
```

```
{ ? }  
procedure matrix_multiply(a: in integer_array_array;  
                           arows, acols: in integer;  
                           b: in integer_array_array;  
                           brows, bcols: in integer;  
                           c: out integer_array_array;  
                           crows, ccols: out integer;);  
  
{ ? }
```

Axiomatic semantics (Hoare logic)

Assignment $P[x/f] \{ x := f \} P$

Consequence
$$\frac{P \rightarrow P' \quad P' \{Q\} R' \quad R' \rightarrow R}{P \{Q\} R}$$

Composition
$$\frac{P \{Q_1\} R_1 \quad R_1 \{Q_2\} R}{P \{Q_1; Q_2\} R}$$

Conditional
$$\frac{(P \wedge B) \{Q_1\} R \quad (P \wedge \neg B) \{Q_2\} R}{P \{\text{if } B \text{ then } Q_1 \text{ else } Q_2\} R}$$

Iteration
$$\frac{(I \wedge B) \{S\} I}{I \{\text{while } B \text{ do } S\} (I \wedge \neg B)}$$

(I is called a *loop invariant*)

Proving program correctness

{P} ← Precondition

Q

{R} ← Postcondition

Strategy: proceed forward from P (or backward from R), and use the rule of consequence

Exercises

```
{ true }
```

```
x := 3;
```

```
{ x = 3 }
```

```
{ x = 3  $\wedge$  y = 5 }
```

```
tmp := x;
```

```
x := y;
```

```
y := tmp;
```

```
{ x = 5  $\wedge$  y = 3 }
```

```
{ true }
```

```
if i < 0
```

```
then absv := - i;
```

```
else absv := i;
```

```
{ absv  $\geq$  0 }
```

Exercises

```
{ true }  
  
i := 1;  
j := N;  
while i ≠ N do  
  j := j - 1;  
  i := i + 1;  
  
{ j = 1 }
```

```
{ true }  
i := 1;  
while i ≤ N do  
  b[N-i+1] := a[i];  
  i := i + 1;  
{ forall i . 1 ≤ i ≤ N : b[N-i+1] = a[i] }
```

array reverse

Exercises

```
{ true }  
i := 2;  
max := a[1];  
while i ≤ N do  
  if a[i] > max then max := a[i];  
  i := i + 1;  
{forall i . 1 ≤ i ≤ N : a[i] ≤ max}
```

max finding

```
{ true }  
i := 1;  
index := -1;  
while i ≤ N ∧ index = -1 do  
  if a[i] = key then index := i;  
  else i := i + 1;  
{ index ≠ -1 ∨ i > N }
```

linear search

Exercises

```
{ true }
```

```
i := 1;  
while i ≤ N do  
  b[i] := a[i] - 1;  
  i := i + 1;
```

```
{ forall i . 1 ≤ i ≤ N : b[i] < a[i] }
```

```
{ true }
```

```
r := x;  
q := 0;  
while y ≤ r do  
  r := r - y;  
  q := q + 1;
```

```
{ y > r ∧ x = r + y * q }
```

*find the quotient q and
remainder r of dividing x by y*

Midterm

Open book/notes, hardcopies only
Emphasis on more recent material (i.e., specification)

Reading for April 30

Chapter 4 (sections 4.1 – 4.2.2)
On the Criteria To Be Used in Decomposing Systems ...