

Program Analysis

(Chapter 6)

Outline

- Analysis vs. testing
- Informal analysis
- Formal analysis: axiomatic semantics

Analysis vs. testing

- Testing characterizes a *single* execution
- Analysis characterizes a *class* of executions; it is based on a *model*
- They have complementary advantages and disadvantages
 - *Soundness*: doesn't miss bugs, might produce false positives
 - *Completeness*: no false positives, might miss bugs

Informal analysis techniques

Code walkthroughs

- Small number of people (three to five)
- Participants: designer, moderator
- Participants receive written documentation from the designer a few days before the meeting
- Predefined duration of meeting (a few hours)
- Focus on the *discovery* of errors, not on fixing them
- Foster cooperation; no evaluation of people
 - Experience shows that most errors are discovered by the designer during the presentation, while trying to explain the design to other people.

Informal analysis techniques

Code inspection

- A reading technique aiming at *discovering common errors*
- Based on checklists; e.g.:
 - use of uninitialized variables;
 - jumps into loops;
 - nonterminating loops;
 - array indexes out of bounds, etc.

Formal analysis: axiomatic semantics (Hoare triples)

A program and its specification (Hoare notation)

```
{true}
begin
  read (a); read (b);
  x := a + b;
  write (x);
end
{output = input1 + input2}
```

proof by backwards substitution

Proof rules

Notation:

If Claim 1 and Claim 2 have been proven,
one can deduce Claim3

$$\frac{\text{Claim1, Claim2}}{\text{Claim3}}$$

Correctness proof

- Partial correctness
 - validity of $\{\text{Pre}\} \text{ Program } \{\text{Post}\}$ guarantees that if the Pre holds before the execution of Program, *and if the program ever terminates*, then Post will be achieved
- Total correctness
 - Pre guarantees Program's termination *and* the truth of Post

An assessment of correctness proofs

- Getting more and more traction in SW development practice
 - proofs become more practical as more powerful support tools are developed
 - Example: Microsoft Spec# (formal specification for C# code)

```
proc Find(xs: [int] int, ct: int, x: int) returns (result: int);  
  requires ct ≥ 0;  
  ensures result ≥ 0 ⇒ result < ct ∧ xs[result]=x;  
  ensures result < 0 ⇒ !(∃ i:int :: 0 ≤ i ∧ i < ct ∧ xs[i] == x);
```

Advantages of correctness proofs

- Verify critical code
 - E.g., French DOE
- Assertions can be the basis for a systematic way of inserting (or even eliminating!) runtime checks
- Knowledge of correctness theory helps programmers being rigorous
 - Pre/post conditions, writing test cases

Examples

{ true }

$y := z + 42;$

$x := z + 50;$

{ $x > y + 7$ }

{ $x > 0 \wedge y > 0$ }

$z := x * y;$

{ $z > -1$ }

Example: The division algorithm

{ true }

$r := x;$

$q := 0;$

while $y \leq r$ do

$r := r - y;$

$q := q + 1;$

{ $y > r \wedge x = r + y * q$ }

*dividing x by y
yields quotient q
and remainder r*

Example: Max finding

```
{ 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}
```

Reading for next class

Sections 6.5 (omit 6.5.2) and 6.7