CSE503: Software Engineering

David Notkin
University of Washington
Department of Computer Science & Engineering
Winter 2002

1

Proving programs correct

- Primary characterization
  - Given a *specification* (in a formal logic) and
  - an *implementation* (in a programming language),
  - prove that the implementation *satisfies* the specification
- Alternative characterization
  - Given the specification,
  - derive (construct) a program that satisfies the specification

{ true }
```
x: int;
read(x);
if (mod(x,2) = 1) then
  x := x + 1;
fi
```
{ even(x) }

2

Key notations for proofs

- The two most common notations are *Hoare triples* and Dijkstra *weakest preconditions* (or predicate transformers)
  - We'll focus primarily on Hoare triples
- A Hoare triple is a logical predicate: **{P}** `S` **{Q}**
- **P** and **Q** are predicates, `S` is a program
- **{P}** `S` **{Q}** is true when
  - if **P** is true, then after `S` executes, **Q** is true

3

Examples

(Note: **X**, **Y** are constants)

- True Hoare triples
  - **{ true }** `y := x*x` **{ y >= 0 }**
  - **{ x > 0 }** `x := x + 1` **{ x > 1 }**
  - **{ x > 1 }** `x := x + 1` **{ x > 0 }**
  - **{ x = X and y = Y }**
    `t := x; x := y; y := t;`
    { x = Y and y = X }
- False Hoare triples
  - **{ true }** `y := x*x` **{ y < 0 }**
  - **{ x = X and y = Y }**
    `t := x; x := y; y := t;`
    { x = Y and t = Y }

4

Another example: true or false?

{ x <> 0 }
```
if x > 0 then
  x := x + 1
else
  x := -x
fi
```
{ x > 0 }

5

Meaning of assignment

- We must precisely define the meaning of the assignment operator used in the programs
- Back-substitution is the basic approach
- Consider the triple **{ P? }** `x := exp` **{ Q(**`x`**) }**
  - **P?** is an unknown precondition
  - **Q** is the postcondition that may be parameterized in terms of the program variable `x`
- For **Q(**`x`**)** to be true requires that P**?** be equal to **Q(**`exp`**)** as a precondition

6

Examples

- **{P? }** `x := x + 1` **{ x > 1 }**
  - **Q**(x) = **x > 1**
  - So **P? = Q**(x+1) = **x + 1 > 1 = x > 0**
- **{ P? }** `y := x*x` **{ y >= 0 }**
  - **Q**(y) = **y >= 0**
  - **So P? = Q**(x*x) = **x*x >= 0 = true**
- This is technically handled by the "proof rule"
  - { **B**[a/X] } `X :=a` { **B** }
  - Where **B[**a/X**]** represents the predicate **B** with all free occurrences of X replaced by a

7

Meaning of conditionals

- There are also proof rules for **{P}** `if C then S1 else S2` **{Q}**
- If we can prove
  - **{P and C}** `S1` **{Q}** and also
  - **{P and not C}** `S2` **{Q}**
- Then we have proven the triple

8

Example

- **{x <> 0}**
  `if x > 0 then x := x + 1 else x := -x`
  {x > 0}
- **(P and C)** = **(x <> 0 and x > 0)**
  = **(x > 0)**
- **{x > 0}** x := x + 1 **{x > 0}** [trivially true]
- **(P and not C)** = **(x <> 0 and not (x > 0)**
  = **(x <> 0) and (x <= 0)**
  = **(x < 0)**
- **{x < 0}** `x := -x` **{x > 0}** [trivially true, QED]

9

Proving programs

- The basic approach to proving a program correct using Hoare triples is to
  - start with the precondition **P**, the postcondition Q, and the program `S`
- `S` usually consists of a sequence of statements
- One then introduces additional intermediate assertions between the statements
  - **{P}** `S1;S2;S3;S4` **{Q}**
  - **{P}** `S1` **{A1}** `S2` **{A2}** `S3` **{A3}** `S4` **{Q}**
- Then prove each triple (they are associative).

10

Some additional proof rules

- What is the semantics of the programming language construct `;`
  - **({P1}**`S1`**{P2}** and **{P2}**`S2`**{P3}) implies {P1}**`S1;S2`**{P3}**
- Also, if **P0 implies P1** in addition, then we also can prove **{P0}**`S1;S2`**{P3}**

11

Loops

- Loops are the biggest challenge in proving correctness, since we can not write simple proof rules because the number of iterations through a loop is in general unbounded
- One issue is proving that the loop terminates; this is usually done separately from the proof about the program's computation
- We have to introduce an added assertion, called a *loop invariant*; it is not generally possible to compute these, so they have to be chosen carefully to allow the proof to go through

12

Termination

- Weak (or partial) correctness: the proof of **{P}**`S`**{Q}** assumes that `S` terminates
- Strong (or total) correctness: Termination of `S` is proven
- Example: weakly correct but not known to be strongly correct

{x > 0}
```
y := f(x);
function f(z : int): int is begin
    if z=1 return 1
    else if even(z) return f(z/2);
    else return f(3*z+1);
end
```
{y = 1}

13

Termination

- It is relatively rare for termination to be the central issue or problem with a program
- Also, demonstrating *non*-termination is equally important for classes of programs, such as operating systems, avionics control systems, etc.

14

Proving loops correct

- **{P}** `while C do S` **{Q}**
- We need to find a loop invariant **I** and prove the following proof obligations
  - **P implies I** // **I** true when the loop starts
  - **{I and C}**`S`**{I}** // I remains true each iteration
  - **(I and not C) implies Q** // if loop terminates, Q holds

15

Simple example (Ghezzi)

- **{x >= 0}** `while x>0 do x := x + 1` **{x = 0}**
- **I = (x >= 0)**
- **P implies I** [trivial]
- **{x >= 0}** `x := x+1` **{x >= 0}** [trivial]
- **((x >= 0) and (x <= 0)) implies (x = 0)** [trivial, QED]
- Question: does this example make sense?

16

Another example: divide `i` by `j`, quotient in `div`, remainder in `t`

{i > 0 and j > 0}
```
t := i;
div := 0;
while t >= j do
    div := div+1;
    t := t-j;
end
```
{i = div*j+t and 0 <= t < j}

In small groups, prove this correct, including explicitly identifying the loop invariant

17

Termination

- Termination is generally proved by the use of well-founded sets
  - A set is well-founded if it is partially ordered and every non-empty subset has a minimal element
- In essence, one wants to show monotonic progress on every iteration towards a fixed bound
- In the previous example, `t` becomes closer to `j-1` on every iteration
  - `while t >= j do t := t-j; end`
- One can ignore the other computations in the loop

18

Next lecture (1/14)

- Weakest precondition formulation
- Proof of correctness of abstract data types

19