

Program Verification

- Deductive argument that a program matches its specification
 - Complementary to testing and inspections
- Predicate logic assertions about program state
 - Values of variables at various program points
- Statement-specific semantic rules
 - Assignment; sequence; conditional; iteration

Program Specification

- The preconditions plus the postconditions together specify *what* it is that the program computes
 - Expressed in terms of only input and output variables
 - Avoids implementation details
 - Assumes termination

Proof that a Program Matches its Specification

- Show that if the program is initiated with the preconditions true, and if it terminates, then the postconditions will also be true
 - Sequence of assertions about program state interleaved with the program's statements
 - Semantic rules for the describing the effect of each kind of statement

Statements

- The proof of a program naturally depends on the program's statements
 - Programming language *semantics*
- We will consider four main types of statements
 - Assignments
 - Blocks
 - Conditionals
 - Iteration statements
- We will also briefly mention subprograms

Assignment Statement

- In program proofs, an assignment statement looks like the following
 - $\{ P \} V \leftarrow E; \{ Q \}$
 - Curly brackets delimit assertions
 - V is the variable being assigned; E is an expression that when evaluated will produce the value assigned to V ; $\{ P \}$ is an assertion describing the state that exists before execution of the assignment; and $\{ Q \}$ is an assertion describing the state afterwards
- In general, we would expect most of the state to be unaffected; only the part that describes V should change
- To describe the semantics of the assignment completely, we need to express the relationship between $\{ P \}$ and $\{ Q \}$

Assignment Statement

- We will consider two cases
- Case 1: V does not occur in E :
 - Just add an assertion about the new value of the variable into the post assertion $\{ Q \}$
 - $\{ P \} \quad X \leftarrow 13; \quad \{ P \wedge (X = 13) \}$
 - Assumes that $\{ P \}$ makes no assertion about X , otherwise we have to remove that assertion from $\{ P \}$
 - $\{ (X = 5) \wedge P \} \quad X \leftarrow 13; \quad \{ P \wedge (X = 13) \}$

Assignment Statement

- Case 2: Left hand side variable appears on right hand side
 - The state before the assignment has a direct contribution to the state after
 - Solve for right hand side variable (the value before the assignment)
 - Substitute solution into pre assertion to get post assertion
 - Use apostrophe to denote value after assignment
 - This is just to distinguish the occurrences of the variable. It can be dropped once the right hand side predicate is determined

Assignment Example

- $\{ P \wedge Q(X) \} \quad X \leftarrow X + Y \quad \{ ? \}$

Assignment Statement

- $\{ P \wedge Q(X) \} \quad X \leftarrow X + Y \quad \{ ? \}$
- Use an apostrophe to distinguish the two occurrences of X

Assignment Statement

- $\{ P \wedge Q(X) \} \quad X \leftarrow X + Y \quad \{ ? \}$
- Use an apostrophe to distinguish the two occurrences of X
- $X' \leftarrow X + Y$

Assignment Statement

- $\{ P \wedge Q(X) \} \quad X \leftarrow X + Y \quad \{ ? \}$
- Use an apostrophe to distinguish the two occurrences of X
- $X' \leftarrow X + Y$
- Solve for X , the value from the state existing before the assignment is executed

Assignment Statement

- $\{ P \wedge Q(X) \} \quad X \leftarrow X + Y \quad \{ ? \}$
- Use an apostrophe to distinguish the two occurrences of X
- $X' \leftarrow X + Y$
- $X = X' - Y$

Assignment Statement

- $\{ P \wedge Q(X) \} \quad X \leftarrow X + Y \quad \{ ? \}$
- $X' \leftarrow X + Y$
- $X = X' - Y$
- Plug into the pre assertion to get the post assertion

Assignment Statement

- $\{ P \wedge Q(X) \} \quad X \leftarrow X + Y \quad \{ ? \}$
- $X' \leftarrow X + Y$
- $X = X' - Y$
- Plug into pre assertion to get post assertion
- $\{ P \wedge Q(X) \} \quad X \leftarrow X + Y \quad \{ P \wedge Q(X' - Y) \}$

Assignment Statement

- $\{ P \wedge Q(X) \} \quad X \leftarrow X + Y \quad \{ ? \}$
- $X' \leftarrow X + Y$
- $X = X' - Y$
- Plug into pre assertion to get post assertion
- $\{ P \wedge Q(X) \} \quad X \leftarrow X + Y \quad \{ P \wedge Q(X' - Y) \}$
- Drop the apostrophe

Assignment Statement

- $\{ P \wedge Q(X) \} \quad X \leftarrow X + Y \quad \{ ? \}$
- $X' \leftarrow X + Y$
- $X = X' - Y$
- Plug into pre assertion to get post assertion
- $\{ P \wedge Q(X) \} \quad X \leftarrow X + Y \quad \{ P \wedge Q(X' - Y) \}$
- Drop the apostrophe
- $\{ P \wedge Q(X) \} \quad X \leftarrow X + Y \quad \{ P \wedge Q(X - Y) \}$

Assignment Example

- $\{ Z = 3 \wedge X > Z \}$ $X \leftarrow X + 4$ $\{ ? \}$

Assignment Example

- $\{Z = 3 \wedge X > Z\} X \leftarrow X + Y \{Z = 3 \wedge (X - 4) > Z\}$

Assignments - General Rule

- $\{ P_E^V \} V \leftarrow E \{ P \}$
- P_E^V is the logical expression obtained from P by substituting the expression E for all occurrences of V
 - Actually, only free occurrences are substituted.

Thus renaming of bound variables may be required

- Example: $\{ a > b \wedge 5 = 5 \} c := 5 \{ a > b \wedge c = 5 \}$
 $\Rightarrow \{ a > b \wedge \text{true} \}$
 $\Rightarrow \{ a > b \}$
- Another example: $\{ a > b + 5 \} b := b + 5 \{ a > b \}$

Sequential Composition

- Consecutive statements can be treated as a single SESE if their assertions can be logically composed

- The essence is to find a single, general predicate that describes the post assertion of the first statement and the pre assertion of the second

- Here is the general rule

$\{P\} S1 \{Q1\}$ and $\{Q1\} S2 \{R\}$ then $\{P\} S1; S2 \{R\}$

- Example

$\{x = 3 \wedge y = 2\} z := x + 2 \{z = 5 \wedge y = 2 \wedge x = 3\} \Rightarrow \{z = 5 \wedge y = 2\}$ and

$\{z = 5 \wedge y = 2\} w := z + y \{z = 5 \wedge y = 2 \wedge w = 7\} \Rightarrow \{z = 5 \wedge w = 7\}$ then

$\{x = 3 \wedge y = 2\} z := x + 2; w := z + y \{z = 5 \wedge w = 7\}$

Conditionals

- Conditionals determine the flow of control based upon the value of some predicate
- The consequent flows (`true` and `false` branches) can assume the corresponding values
 - That is, the `true` arm can assume that the values of the variables make the predicate `true`. Contrariwise, for the `false` arm

Conditional Example

- $\{\text{true}\}$ if B then $x := 3$ else $y := 4$ $\{?\}$

Conditional Example

- $\{\text{true}\}$ if B then $x := 3$ else $y := 4$ $\{x = 3 \vee y = 4\}$

Another Conditional Examples

$\{\text{true}\}$ if $x > 0$ then $y := x$ else $y := -x$ $\{?\}$

Another Conditional Examples

$\{\text{true}\}$ if $x > 0$ then $y := x$ else $y := -x$ $\{?\}$

- Approach: treat each arm separately, then generalize the results

Another Conditional Examples

- $\{\text{true}\}$ if $x > 0$ then $y := x$ $\{x > 0 \wedge y = x\}$
 else $y := -x$ $\{x \leq 0 \wedge y = -x\}$

Another Conditional Examples

- $\{\text{true}\}$ if $x > 0$ then $y := x$ $\{x > 0 \wedge y = x\} \Rightarrow \{y = \text{abs}(x)\}$
else $y := -x$ $\{x \leq 0 \wedge y == -x\} \Rightarrow \{y = \text{abs}(x)\}$

Another Conditional Examples

- $\{\text{true}\}$ if $x > 0$ then $y := x$ $\{x > 0 \wedge y = x\} \Rightarrow \{y = \text{abs}(x)\}$
 else $y := -x$ $\{x \leq 0 \wedge y == -x\} \Rightarrow \{y = \text{abs}(x)\}$
- $\{\text{true}\}$ if $x > 0$ then $y := x$ else $y := -x$ $\{y = \text{abs}(x)\}$

Iteration

- An assertion inside a loop must have three properties
 1. It must be true first time the loop is executed
 2. It must be true for a typical execution (assuming it was true the last time)
 3. It must be *strong enough* to imply what is needed to verify the rest of the program
- Such assertions are called *loop invariants*
- Termination of the loop must be demonstrated separately

Loop Invariants

- A loop typically executes multiple times
- A loop invariant is an assertion that holds each time (*i.e.* it is invariant)
- This means that a loop invariant is typically expressed in terms of the progress that the loop has made so far (*i.e.* in terms of the loop's index variable)
- Coming up with a loop invariant requires insight into the purpose of the loop

Exercise

```
int b[0 : n]; i := 1; t := b[0];
```

```
while i <= n do
```

```
  t := t + b[i];
```

```
  i := i + 1
```

```
end;
```

<----- What is always true here?

Exercise

```
int b[0 : n]; i := 1; t := b[0];
```

```
while i <= n do
```

```
  t := t + b[i];
```

```
  i := i + 1
```

```
end;
```

<----- $\{1 \leq i \leq n \wedge t = \sum_{k=0}^{i-1} b[k]\}$

Solution

```
int b[0 : n]; i := 1; t := b[0];
```

```
while i <= n do
```

```
  t := t + b[i];
```

```
  i := i + 1
```

```
end;
```

----- $\{n > 0\}$

----- $\{n > 0 \wedge i = 1 \wedge t = b[0]\}$

----- $\{1 \leq i \leq n \wedge t = \sum_{k=0}^{i-1} b[k]\}$

----- $\{1 \leq i \leq n \wedge t = t' + b[i] =$
 $\sum_{k=0}^{i-1} b[k] + b[i] = \sum_{k=0}^i b[k]\}$

----- $\{1 \leq i - 1 \leq n \wedge t = \sum_{k=0}^{i-1} b[k]\}$

----- $\{i > n \wedge t = \sum_{k=0}^n b[k]\}$

Subprograms

- Programs frequently make use of subprograms. Hence, program proofs must somehow deal with the correctness of their subprograms
- A subprogram can have pre and postconditions just like a program. Hence, it can be treated like any other statement
 - But how about recursive calls?

Procedure Calls

$$\{ \mathbf{Q}^{x_1, \dots, x_m, y_1, \dots, y_n} \}_{f_1(x,y), \dots, f_m(x,y), g_1(x,y), \dots, g_n(x,y)}^{f(x)} \{ \mathbf{Q} \}$$

- x is the list of actual parameters
- $x_1 \dots x_m$ are those elements of x that correspond to var parameters
- y is the list of variables accessed non-locally
- y_1, \dots, y_n are elements of y assigned inside of p
- $x_1, \dots, x_m, y_1, \dots, y_n$ are distinct, otherwise the effect of p is undefined.
This is the *aliasing* problem of two names referring to the same value
- $f_i(x, y)$ maps the initial value of x_i to its final value.
- $g_j(x, y)$ maps the initial value of y_j to its final value
- All functions are computed simultaneously

Procedure Call Examples

CALLING CONTEXT	FUNCTION	RESULTING STATE
{true} x := f()	f: return 3;	{x = 3}
{y = 4} x := f(y)	f(z): return z + 2;	{y = 4 ∧ x = 6}
{true} f(y)	f(z): var z; z := 3;	{y = 3}
{true} f(x, y)	f(z, w): var z, w; z := 3; w := 2;	{x = 3 ∧ y = 2}
{true} f()	f(): a := 3;	{a = 3}
{true} f()	f(): a := 3; b := 4;	{a = 3; b = 4}

Outstanding Issues

- Computer arithmetic
 - Formal verification usually assumes mathematical laws
 - Algebraic simplifications
- Rules of inference
- Procedure calls
- Termination
- Object-oriented programs
 - Treat instance variables as if they were globals (available for use in pre/post conditions)
- Concurrent programs
 - Must deal with shared variables accessed by separate threads

Proof Mechanics

- In constructing an actual proof you can make use of two other kinds of rules
 - Arithmetical simplifications
 - Rules of inference

Example Rule of Inference

- *Rules of consequence:*

$$\{ P \} S \{ Q \} \text{ and } \{ Q \} \Rightarrow \{ R \} \text{ then } \{ P \} S \{ R \}$$
$$\{ P \} S \{ Q \} \text{ and } \{ R \} \Rightarrow \{ P \} \text{ then } \{ R \} S \{ Q \}$$

- For statement S and predicates P, Q, and R

- That is, you can *weaken* post-conditions and *strengthen* preconditions

Termination

- Proved independently from correctness
- The only place where termination is an issue is the loop
- Each loop is shown independently to terminate, from the inside out
- Normally handled by proving that the loop makes discernable progress on every iteration

Termination Method

- Introducing an auxiliary function (f)
- f 's domain is the set of program variables; its range is the non-negative integers
- f is shown to be monotonically decreasing. That is, every time through the loop its value decreases
- f is defined in such a way that the loop cannot execute when f is non-positive
- Thus, if f ever reaches zero, the loop must stop