Formal Verification

Prof. Leon Osterweil
Computer Science 520/620
Spring 2014
Some Examples of “Relations”

• Executing this code must compute this function
• This code must conform to that design element
• This compiled code came from this compiler
• This design element addresses those requirements
• These lower level requirements are elaborations of these higher level requirements
• This is the date by which that test must be passed
• Component invocations conform to component abstract interface specifications
• Documentation describes the actual system
• ETC.....
Evaluation of Static Analysis

- **Strengths:**
  - Can demonstrate the absence of faults
  - Proofs can be automatically generated and proven
  - Algorithms are fast (low-order polynomial)
  - No need to generate test data
  - You know when you are done

- **Weaknesses**
  - Behavior specification is a model with inaccuracies
    - Not all paths are executable
  - Only certain classes of faults analyzable
    - Mostly sequence specific
    - Weak on functionality
Formal Verification

- Prove that a program must always deliver the specified functional behavior
  - Regardless of what path is executed
  - Even if there are unlimited numbers of paths
- Requires a formal specification of the desired functional behavior.
- Requires a formal specification of the computation that is to deliver this behavior
- Requires solid mathematical reasoning

Previously this was misleadingly called “Proof of Correctness”

Formal Verification

- Intent
  - Usually specification of functionality
  - What function(s) does the software compute?
  - Sometimes accuracy, timing, ...
- Behavior
  - Inferred from semantically rich program model
  - Generally requires most of semantics of programming language
  - Generally uses symbolic execution
- Comparison
  - Use of formal mathematics (eg. predicate logic)
Formal Verification

Specification of Intended Behavior

Final and Intermediate First-Order Logic Assertions

Symbolic Execution of Path Segments

Comparison of Behavior to Intent

First-Order Logic Theorems and Proofs

Not necessarily functionality, could also be timing, etc.

Not necessarily code segments

Proof of the Absence of All Functional Faults

First-Order Logic

Theorems and Proofs

Specification of Actual Behavior

Comparison of Behavior to Intent
Floyd Method of Inductive Assertions

- Show that each program fragment behaves as intended
- Use induction to prove that all sequences of executable fragments behave as intended
- Show that the program must terminate

Example Assertion

```plaintext
X := Y;
Time := Y * 2.0 * T;
ASSERT Time > 0.0;
```

<rest of code>
Use of Assertions in Testing

• Assertions: Specifications of intended relations among the values of program variables
• Comparison: Runtime evaluation of assertions
  – Facilities for programming reactions to violations
• Examine behaviors at internal program locations while the program is executing
  – Augments examining only final outputs
• Previously seen as a debugging aid
• Now we will see it as a key support for static verification

<code sequence>
X := Y;
Time := Y * 2.0 * T;
**ASSERT Time > 0.0;**
<rest of code>

Inserted by Tester

Automatically processed into

if ~(Time > 0.0) Then
  Assertion_violation_handler;

Use of Assertions in Verification

- **Assertion**: Specification of a condition that is intended to be true at a specific given site in the program text
- Floyd's Method assertions are written in Predicate Logic
- In Floyd's Method there are three types of assertions
  - Initial, A₀: Sited at the program initial statement
  - Final, Aₙ: Sited at the program final statement
  - Intermediate Aᵢ: Often called a "loop invariants"
- Sited at various program locations subject to the rule:
  
  EVERY LOOP ITERATION (CFG CYCLE) SHALL PASS THRU THE SITE OF AT LEAST ONE INTERMEDIATE ASSERTION

Net Effect: Every program execution sequence is divided into a finite number of segments of non-looping code bounded on each end by a predicate logic assertion

---

Mathematical Induction

- **Goal**: prove that a given property holds for all elements of a set
- **Approach**:
  - (initial step) show property holds for "first" element
  - (induction step) show that if property holds for element i, then it must also hold for element i + i
- Often used when direct analytic techniques are too hard or complex
Example: How many edges in $C_n$

Theorem:

Let $C_n = (V_n, E_n)$ be a complete, unordered graph on $n$ nodes,

then $|E_n| = n \times (n-1)/2$

Initial Step

- show the property is true for $C_1$:
- graph has one node, 0 edges

$|E_1| = n(n-1)/2 = 1(0)/2 = 0$
**Induction Step**

- assume true for $C_n$: $|E_n| = n(n-1)/2$
- graph $C_{n+1}$ has one more node, but $n$ more edges (one from the new node to each of the $n$ old nodes)
- so, $|E_{n+1}| = n(n-1)/2 + n$
  - $= n(n-1)/2 + 2n/2 = (n(n-1)+2n)/2$
  - $= (n(n-1+2))/2 = n(n+1)/2$
  - $= (n+1)((n+1)-1)/2$
  - $= (n+1)(n)/2$

**Floyd’s Method of inductive assertions (informal description)**

- Place assertions at the start, final, and intermediate points in the code.
- Any path is composed of sequences of program fragments all of which:
  - start with an assertion
  - are followed by some code that is:
    - Assertion-free
    - Loop-free
  - and end with an assertion
  eg. $(A_s = A_0, C_1, A_1, C_2, A_2,...A_{n-1}, C_{n-1}, A_n, C_n, A_{n+1}(=A_f))$
- Show that for any executable path, if $A_i$ is assumed true for any $A_i$ and code $C_i$ is executed, then $A_{i+1}$ must always be true
Induction in Floyd’s Method

- Initially
  - Verify that the Initial Assertion is correct
- Induction on length of execution path
- Prove that function computed by all execution paths of length \( L + 1 \) are correct
  - Provided that all execution paths of length \( L \) compute the correct function
- Inductive step is the hard one (as usual)
- Proof relies on the fact that there are only a (relatively) small number of path segments in any program.

Pictorially

- Initial assertion
- Intermediate assertions
- Final assertion

STRAIGHT-LINE CODE

\( A_i \) \( \quad C_i \quad \) \( A_{i+1} \)
Must be sure:

assuming $A_i$, then executing Code $C_i$, necessarily $\Rightarrow A_{i+1}$

by forward substitution

\[
\begin{array}{c}
A_i \\
\hline
C_i \\
\hline
A_{i+1}
\end{array}
\]

STRAIGHT-LINE CODE

KEY POINT: There are only a small number of these in any realistic program
Why does this work?

suppose \( P = \) arbitrary path through the program
can denote it by
\[
P = A_s \ C_1 \ A_1 \ C_2 \ A_2 \ldots C_n \ A_f
\]
where
- \( A_s \) - Initial assertion
- \( A_f \) - Final assertion
- \( A_i \) - Intermediate assertions
- \( C_i \) - Loop free, uninterrupted, straight-line code

If it has been shown that
\[
\forall \ i, \ 1 \leq i < n: A_i \ C_i \Rightarrow A_{i+1}
\]
Then, by induction
\[
A_s \ldots \Rightarrow A_f
\]
AGAIN: There are only a small number of these in any realistic program

Loop Invariants
(Loop Breakers)

- Problem: infinite number of paths
  - Must find a way to deal with loops
- Solution: Assertion, \( A_i \), that is
  - True for any number of loop iterations
  - “connects up” to adjacent assertions
- Such an assertion:
  - Is invariant with respect to loop iterations
  - Must be embedded in (break) every loop

A loop invariant must capture the essence
Of the work that the loop is to perform
Schematic Example of a Loop Invariant

Aᵢ is a loop invariant because of its relation to other assertions:

NOTE THAT:
Aᵢ, false branch,  => Aᵢ
Aᵢ, true branch,  => Aᵢ

BUT ALSO:
Initial assertion Aₛ to Aᵢ => Aᵢ
Aᵢ, false branch,  => final assertion Aᵢ
Aᵢ, true branch,  => final assertion Aᵢ

How Many Path Segments?

Aₛ -> Aᵢ
Aᵢ true -> Aᵢ
Aᵢ false -> Aᵢ
Aᵢ true -> Aᵢ
Aᵢ false-> Aᵢ
Aₛ -> Aᵢ
Floyd’s Method (carefully stated)

- Specify initial, final assertions to capture intent
- Intermediate assertions "cut" every program loop
- For each pair of assertions with an executable (assertion-free) path from the first to the second,
  - Assume that the first assertion is true
  - Show that for all (assertion-free, executable) paths from the first assertion to the second, that the second assertion is true
- This establishes “partial correctness”
- Show that the program terminates
  - This establishes “total correctness”

A Little-JIL Definition of Floyd’s Method of Inductive Assertions
Rework Steps—Implemented by Recursion
Parameters passed include history of all exceptions thrown and how they were handled.

Sort Example

procedure sort(values, size);
declare values real array[1000], temp real,
i, j, size integer;

ASSERT INITIAL
do for i = 1 to size-1
  do for j = i+1 to size
    if values[j] > values[i] then
      temp := values[i];
      values[i] := values[j];
      values[j] := temp;
  end do;
end do;

ASSERT INNER
end do;

ASSERT OUTER
end do;

ASSERT FINAL
end sort;
Sort Example

procedure sort(values, size);
declare values real array[1000], temp real,
i, j, size integer;

ASSERT INITIAL
do for i = 1 to size-1
do for j = i+1 to size
if values[j] > values[i] then
    temp := values[i];
    values[i] := values[j];
    values[j] := temp;

ASSERT INNER
end do;

ASSERT OUTER
end do;

ASSERT FINAL
end sort;

ASSERT OUTER

values[i] ≥ values[j] for all j, where i < j ≤ size
Sort Example

procedure sort(values, size);
declare values real array[1000], temp real,
   i, j, size integer;

ASSERT INITIAL
do for i = 1 to size-1
   do for j = i+1 to size
      if values[j] > values[i] then
         temp := values[i];
         values[i] := values[j];
         values[j] := temp;
   end do;
end do;

ASSERT OUTER
end do;

ASSERT FINAL
end sort;

ASSERT INNER

values[i] ≥ values[k] for all i < k ≤ j
Sort Example

procedure sort(values, size);
declare values real array[1000], temp real, 
i, j, size integer;

ASSERT INITIAL
do for i = 1 to size-1
  do for j = i+1 to size
    if values[j] > values[i] then
      temp := values[i];
      values[i] := values[j];
      values[j] := temp;

ASSERT INNER
end do;
ASSERT OUTER
end do;
ASSERT FINAL
end sort;

ASSERT INITIAL

size > 0

and

size < 1000
Sort Example

procedure sort(values, size);
    declare values real array[1000], temp real,
    i, j, size integer;

    ASSERT INITIAL
    do for i = 1 to size-1
        do for j = i+1 to size
            if values[j] > values[i] then
                temp := values[i];
                values[i] := values[j];
                values[j] := temp;
            end if;
        end do;
    end do;

    ASSERT OUTER
    end do;

    ASSERT FINAL
end sort;

ASSERT FINAL

For all i < j ≤ size:
    values[i] ≥ values[j]
ASSERT FINAL

For all $i < j \leq \text{size}$:
\[ \text{values}[i] \geq \text{values}[j] \]

AND: values’ is a permutation of values:

Lemma 1: ASSERT INNER, false $\Rightarrow$ ASSERT INNER

procedure sort(values, size);
declare values real array[1000], temp real,
i, j, size integer;

ASSERT INITIAL
do for $i = 1$ to size-1
  do for $j = i+1$ to size
    if values[$j$] > values[$i$] then
      temp := values[$i$];
      values[$i$] := values[$j$];
      values[$j$] := temp;
    end if
  end do;
end do;

ASSERT OUTER
end do;

ASSERT FINAL
end sort;
Lemma 1: ASSERT INNER, false => ASSERT INNER

procedure sort(values, size);
declarare values real array[1000], temp, i, j, size integer;

ASSERT INITIAL
do for i = 1 to size-1
do for j = i+1 to size
if values[j] > values[i] then
    temp := values[i];
    values[i] := values[j];
    values[j] := temp;
end do;

ASSERT OUTER
end do;

ASSERT FINAL
end sort;

Symbolic Execution:

values[i] ≥ values[k] for all i < k ≤ j-1
Lemma 1: ASSERT INNER, false => ASSERT INNER

procedure sort(values, size);
declare values real array[1000], temp real,
i, j, size integer;

ASSERT INITIAL
do for i = 1 to size-1
do for j = i+1 to size
  if values[j] > values[i] then
    temp := values[i];
    values[i] := values[j];
    values[j] := temp;
end do;

ASSERT OUTER
end do;

ASSERT FINAL
end sort;

Symbolic Execution:

Symb. Vals.              Path Conditions
values[j] = IN_1
values[i] = IN_2

values[j] ≤ values[i]
Lemma 1: ASSERT INNER, false => ASSERT INNER

procedure sort(values, size);
declare values real array[1000], temp real,
i, j, size integer;

ASSERT INITIAL
do for i = 1 to size-1
do for j = i+1 to size
if values[j] > values[i] then
    temp := values[i];
    values[i] := values[j];
    values[j] := temp;
end do;
end do;

ASSERT OUTER
end do;

ASSERT FINAL
end sort;

ASSERT INNER:
values[i] ≥ values[k] for all i < k ≤ j-1

Symbolic Execution:

Symb. Vals.     Path Conditions
values[j] = IN1  values[j] ≤ values[i]
values[i] = IN2  IN1 ≤ IN2

Thus:
values[i]' ≥ values[j]' (because IN1 ≤ IN2)

values[i]' ≥ values[k]' for all i < k ≤ j
Lemma 1: \textbf{ASSERT INNER, false $\Rightarrow$ ASSERT INNER}

```
procedure sort(values, size);
  declare values real array[1000], temp real,
    i, j, size integer;

  ASSERT INITIAL
  do for i = 1 to size-1
    do for j = i+1 to size
      if values[j] > values[i] then
        temp := values[i];
        values[i] := values[j];
        values[j] := temp;
      end do;
    end do;
  end do;

  ASSERT OUTER
  end do;

  ASSERT FINAL
  end sort;
```

**ASSERT INNER:**
values[i] $\geq$ values[k] for all $i < k \leq j$-1

**Symbolic Execution:**

<table>
<thead>
<tr>
<th>Symb. Vals.</th>
<th>Path Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>values[j] = IN</td>
<td>values[j] $\leq$ values[i]</td>
</tr>
<tr>
<td>values[i] = IN</td>
<td>values[i] = IN</td>
</tr>
<tr>
<td>IN_1 $\leq$ IN_2</td>
<td></td>
</tr>
</tbody>
</table>

Thus:
values[i]’ $\geq$ values[j]’
(because IN_1 $\leq$ IN_2) and so

**ASSERT INNER:**
values[i]’ $\geq$ values[k]’ for all $i < k \leq j$

Lemma 2: \textbf{ASSERT INNER, true $\Rightarrow$ ASSERT INNER}

```
procedure sort(values, size);
  declare values real array[1000], temp real,
    i, j, size integer;

  ASSERT INITIAL
  do for i = 1 to size-1
    do for j = i+1 to size
      if values[j] > values[i] then
        temp := values[i];
        values[i] := values[j];
        values[j] := temp;
      end do;
    end do;
  end do;

  ASSERT OUTER
  end do;

  ASSERT FINAL
  end sort;
```
Lemma 2: ASSERT INNER, true => ASSERT INNER

procedure sort(values, size);
declare values real array[1000], temp real,
i, j, size integer;
ASSERT INITIAL
do for i = 1 to size-1
  do for j = i+1 to size
    if values[j] > values[i] then
      temp := values[i];
      values[i] := values[j];
      values[j] := temp;
  end do;
end do;
ASSERT OUTER
end do;
ASSERT FINAL
end sort;
Lemma 2: ASSERT INNER, true => ASSERT INNER

procedure sort(values, size);
declare values real array[1000], temp real,
i, j, size integer;

ASSERT INITIAL
do for i = 1 to size-1
do for j = i+1 to size
    if values[j] > values[i] then
        temp := values[i];
        values[i] := values[j];
        values[j] := temp;
end do;

ASSERT INNER:
values[i] ≥ values[k]  for all i < k ≤ j-1

Symbolic Execution:
Symb. Vals.       Path Conditions
values[j] = IN_1
values[i] = IN_2

ASSERT OUTER
end do;

ASSERT FINAL
end sort;
Lemma 2: ASSERT INNER, true => ASSERT INNER

procedure sort(values, size);
declare values real array[1000], temp real, i, j, size integer;

ASSERT INITIAL
do for i = 1 to size-1
do for j = i+1 to size
  if values[j] > values[i] then
    temp := values[i];
    values[i] := values[j];
    values[j] := temp;
  end do;

ASSERT INNER:
values[i] ≥ values[k] for all i < k ≤ j-1

Symbolic Execution:
Symb. Vals. Path Conditions
values[j] = IN values[j] > values[i] IN1 > IN2
values[i] = IN2
temp = IN2

ASSERT OUTER
end do;

ASSERT FINAL
end sort;
Lemma 2: ASSERT INNER, true \(\Rightarrow\) ASSERT INNER

procedure sort(values, size);
declare values real array[1000], temp real,
i, j, size integer;

ASSERT INITIAL
do for i = 1 to size-1
do for j = i+1 to size
if values[j] > values[i] then
  temp := values[i];
  values[i] := values[j];
  values[j] := temp;
end do;

ASSERT INNER
values[i] \geq values[k] \text{ for all } i < k \leq j-1

Symbolic Execution:
Symb. Vals.       Path Conditions
values[j] = IN\_2     values[j] > values[i]
values[i] = IN\_1
temp = IN\_1

ASSERT OUTER
end do;

ASSERT FINAL
end sort;
Lemma 2: ASSERT INNER, true => ASSERT INNER

procedure sort(values, size);
declare values real array[1000], temp real,
i, j, size integer;

ASSERT INITIAL
for i = 1 to size-1
do for j = i+1 to size
if values[j] > values[i] then
    temp := values[i];
    values[i] := values[j];
    values[j] := temp;
end do;
end do;

ASSERT OUTER
end do;

ASSERT FINAL
end sort;

Symbolic Execution:

<table>
<thead>
<tr>
<th>Symb. Vals.</th>
<th>Path Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>values[j] = IN₁</td>
<td>values[j] &gt; values[i]</td>
</tr>
<tr>
<td>values[i] = IN₂</td>
<td>IN₁ &gt; IN₂</td>
</tr>
</tbody>
</table>

Thus:
values[i]' > values[j]' (because IN₁ > IN₂)

values[i] ≥ values[k]' for all i < k ≤ j
Lemma 2: ASSERT INNER, true => ASSERT INNER

procedure sort(values, size);
declare values real array[1000], temp real,
i, j, size integer;

ASSERT INITIAL
do for i = 1 to size-1
do for j = i+1 to size
if values[j] > values[i] then
    temp := values[i];
    values[i] := values[j];
    values[j] := temp;
end do;

ASSERT INNER

end do;

ASSERT OUTER
end do;

ASSERT FINAL
end sort;

ASSERT INNER:
values[i] ≥ values[k] for all i < k ≤ j-1

Symbolic Execution:

Symb. Vals.          Path Conditions
values[j] = IN1  values[j] > values[i]
values[i] = IN2   IN1 > IN2

Thus:
values[i]' > values[j]'
(because IN1 > IN2) and so

ASSERT INNER:
values[i]' ≥ values[k]' for all i < k ≤ j

NOTE: Proving that values' is a permutation of values derives from proving this is true for this step.
Loop Invariants

• Should be able to identify loop invariant for every loop that you write
• Should be able to state initial and final assertions too
• Maybe not in formal mathematics
• Should be able to reason about how they support overall program logic

An Example:
Wensley's Algorithm

Procedure Wensley (P: input, Q: input, E: input, Y: output);
--assume 0≤ P<Q, 0< E
Declare P, Q, E, Y, A, B, D real;
A :=0.0; B :=Q / 2.0; D :=1.0; Y := 0.0;
Do_While (D>=E)
    If (P - A - B >= 0.0) then {Y := Y+(D / 2.0); A := A+B};
    B :=B / 2.0; D := D / 2.0;
End_do;
End Wensley;
Wensley’s Algorithm

Input P, Q, E

A ← 0.0
B ← Q/2
D ← 1.0
Y ← 0.0

D ≥ E

P-A-B ≥ 0.0

Y ← Y+(D/2.0)
A ← A+B

B ← B/2.0
D ← D/2.0

What does Wensley's algorithm do?

• Approximating P/Q (=Y) with error ≤ E
• On the kth iteration of the loop

A_k = a_1 Q2^{-1} + a_2 Q2^{-2} + ... + a_k Q2^{-k}
 a_i ∈ {0,1}
B_k = Q2^{-k}
D_k = 2^{-k}
Y_k = a_1 2^{-1} + a_2 2^{-2} + ... + a_k 2^{-k}
 a_i ∈ {0,1}
What does Wensley's algorithm do?

- since $0 \leq P/Q < 1$, then $P/Q$ can be estimated as a sum of the series
  
  $$a_12^{-1} + a_22^{-2} + ... + a_k2^{-k}$$

  $$a_i \in \{0, 1\}$$

- therefore
  
  - $Y_k$ is the computed value of the quotient
  - given $Y_k$, $A_k$ is the computed dividend $P$
  - $D_k$ is the computed error
  - $P-(A_k + B_k)$ says when to add $2^{-(k+1)}$ to $Y_{k+1}$ ($a_{k+1}=1$)

\[ B_k = 2^{-(k+1)} * Q \]

Wensley’s

$$P/Q \rightarrow 0.01011110101011110000101010101010101010101010...$$
Wensley's

\[ \frac{P}{Q} \rightarrow 0.010111101010111100001010101010\ldots \]

Wensley's

\[ \frac{P}{Q} \rightarrow 0.010111101010111100001010101010\ldots \]

\[ Y_k \]
Assertions

Initial: \( A_s: \{(0 \leq P < Q) \land (0 < E)\} \)

Final: \( A_f: \{((P/Q-E) < Y \leq (P/Q))\} \)

Intermediate:
\[
A_i: \{(A = Q \times Y) \land (B = Q \times (D/2)) \land (k \geq 0, k \text{ integer} \land D = 2^{-k}) \land ((P/Q) - D) < Y \leq (P/Q)\}
\]

Summary of Four Lemmas Needed

I. Initial assertion, \( A_0 \), to \( A_i \)
II. \( A_i \), false branch, to \( A_i \)
III. \( A_i \), true branch, to \( A_i \)
IV. \( A_i \), to \( A_f \), final assertion

Lemmas called verification conditions
Lemma I: $A_s$ to $A_i$

$A_s$: Initial Assertion

$(0 \leq P < Q) \land (0 < E)$

Input $P, Q, E$

$A \leftarrow 0$
$B \leftarrow Q/2$
$D \leftarrow 1$
$Y \leftarrow 0$

$A_i$: $A = Q \times Y$
$B = Q \times D/2$
$D = 2^{-k}$, $k = 0$

$P/Q - D < Y \leq P/Q$

Proof:
1) $A = 0$; $Q \times Y = Q \times 0 = 0$
   $A = Q \times Y$
2) $B = Q/2 = Q \times 1/2 = Q \times D/2$
3) $D = 1 = 2^0$
4) $P/Q - 1 < Y = 0 \leq P/Q$

because $0 \leq P/Q < 1$ given by input assertion
Lemma II: $A_i$, false branch, $A_i$

$A_i$:  
- $A = Q \cdot Y$  
- $B = Q \cdot D/2$  
- $D = 2^k$ for some integer $k$  
- $P/Q - D < Y \leq P/Q$

- $D \geq E$ [constraint]  
- $P - A - B < 0$ [constraint]  
- $B \leftarrow B/2$  
- $D \leftarrow D/2$

$\Rightarrow A_i$:  
- $A = Q \cdot Y$  
- $B = Q \cdot D/2$  
- $D = 2^{-(k+1)}$  
- $P/Q - D < Y \leq P/Q$

Proof of lemma II

- Need to establish that $A_i$ is a correct relation among variable values after loop execution, based on assumption that $A_i$ was correct among variable values before loop execution

- Notation:
  - $A$, $B$, $D$, $Y$ are original values of variables  
  - $A'$, $B'$, $D'$, $Y'$ are values after loop execution

- Code execution gives:
  - $A' = A$, $B' = B/2$; $D' = D/2$; $Y' = Y$
Proof of Lemma II

\[ A = Q \times Y \]
\[ B = Q \times D/2 \]
\[ D = 2^k \text{ for some integer } k \]
\[ P/Q - D < Y \leq P/Q \]

D \geq E \text{ [constraint]}
P - A - B < 0 \text{ [constraint]}
B \leftarrow B/2
D \leftarrow D/2

\[ A' = Q \times Y' \]
\[ B' = Q \times D'/2 \]
\[ D' = 2^{(k+1)} \text{ for some integer } k \]
\[ P/Q - D' < Y' \leq P/Q \]

(Code execution shows
\[ A' = A, B' = B/2; D' = D/2; Y' = Y \]

Proof:
1) \[ A' = A = Q \times Y = Q \times Y' \]
2) \[ B' = B/2 = (Q \times D/2)/2 \]
\[ = (Q \times 2D'/2)/2 = Q \times D'/2 \]
3) \[ D' = D/2 = 2^{-k}/2 = 2^{-k-1}=2^{-(k+1)} \]
4) a)P-A-B < 0 \text{ (constraint) }
\[ P - Q \times Y - Q \times D/2 < 0 \]
\[ P/Q - Y - D/2 < 0 \text{ (uses } Q > 0 \text{) } \]
\[ P/Q - D/2 < Y \]
\[ \text{but } D' = D/2, \text{ so } P/Q - D' < Y' \]
b) \[ Y \leq P/Q \Rightarrow Y' \leq P/Q \]

Lemma III: A\(i\); True branch; A\(i\)

A: \[ A = Q \times Y \]
\[ B = Q \times D/2 \]
\[ D = 2^k \text{ for some integer } k \]
\[ P/Q - D < Y \leq P/Q \]

D \geq E \text{ [constraint]}
P - A - B \geq 0 \text{ [constraint]}
Y \leftarrow Y + (D/2.0)
A \leftarrow A + B
B \leftarrow B/2.0
D \leftarrow D/2.0

From symbolic execution we know:
\[ A'' = A + B; \quad B'' = B / 2; \]
\[ D'' = D / 2; \quad Y'' = Y + D / 2; \]

We also know: P - A - B \geq 0 and D \geq E
Proof Lemma III

\begin{align*}
A_1: & \quad A = Q \times Y \\
B = & \quad Q \times D/2 \\
D = & \quad 2^k \text{ for some integer } k \\
P/Q - D < Y \leq P/Q \\
\end{align*}

From symbolic execution we know:

\begin{align*}
A'' = & \quad A + B \quad B'' = B / 2 \\
D'' = & \quad D / 2 \quad Y'' = Y + D / 2 \\
\end{align*}

We also know: \( P - A - B \geq 0 \) and \( D \geq E \)

Proof:

1) \( A'' = A + B = Q \times Y + Q \times (D/2) = Q \times Y' \\
2) \( B'' = B/2 = Q \times D/2/2 = Q \times D' /2 \\
3) \( D'' = D/2 = 2^{-(k+1)} = 2^{-(k+1)} \\
\begin{align*}
4) & \quad a) \quad P - A - B \geq 0 \Rightarrow P - Q \times Y - Q \times (D/2) \geq 0 \\
& \quad \Rightarrow P/Q - D/2 \geq Y \Rightarrow P/Q - D/2 \geq Y'' - D/2 \\
& \quad \Rightarrow P/Q \geq Y'' \\
& \quad b) \quad P/Q - D < Y \Rightarrow P/Q - D < Y'' - D/2 \\
& \quad \Rightarrow P/Q - D/2 < Y'' \Rightarrow P/Q - D'' < Y'' \\
\end{align*}

Lemma IV \( A_i, A_f \)

- \( A_i \), false, \( A_f \)
Lemma IV

\[ A_i : (A=Q*Y) \land (B=Q*(D/2)) \land (k\geq0, k \text{ integer} \land D=2^{-k}) \land ((P/Q)-D)<Y\leq(P/Q) \]

\[ D < E \] code

\[ \Rightarrow A_f : ((P/Q)-E)<Y\leq(P/Q)) \]

Proof:
Given \(((P/Q)-D)<Y\leq(P/Q)\) and \((D < E) \Rightarrow ((P/Q)-E)<((P/Q)-D)<Y\leq(P/Q) \Rightarrow A_f\)

This is only partial correctness

- Must also prove termination
  - In general, can not prove termination
  - For most programs, can usually do it by showing that each loop must terminate

- For our example:
given that \((E>0)\) observe that \(D\) is halved on each iteration and \(E\) does not change
Thus, eventually \(D<E\) and the loop terminates
Observations

- Proofs are long, tedious & often hard
- Assertions are hard to get right
- Invariants are difficult to get right.
  - need to be invariant, but also need to support overall proof strategy
- Proofs themselves often require deep program insight
  - Often require axioms about the domain

Deeper Issues

- Unsuccessful proof attempt $\Rightarrow$ ???
  - incorrect software
  - incorrect assertions
  - incorrect placement of assertions
  - inept prover
  - any combination (or all) of the above
- Although failed proofs often indicate which of the above is likely to be the problem (especially to an astute prover)
Deeper Issues

- Undecidability of Predicate calculus -- no way to be sure when you have a false theorem
  - There is no sure way to know when you should quit trying to prove a theorem (and change something)
- Proofs are generally much longer than the software being verified
  - Suggests that errors in the proof are more likely than errors in the software being verified

Mathematics as a "social process"

- Belief in a proof is a social process
  - Informally describe proof
  - Distribute an informal write-up to colleagues
  - Formal write-up is refereed
  - Accepted paper gets read by wider audience
  - Proof/Theorem is used
  - Increases confidence
- Despite this, mathematical proofs are often wrong
Specification Problem

• Real programs are not captured by simple mathematical algorithms
  – E.g. “This software correctly identifies faces”
  – Error processing issues
  – User interface issues
• Resulting specifications are
  – Large
  – Mathematically unappealing
  – Probably not complete
  – Hard to capture intent

Specification Problem

• Specification & program are not independent representations
  – Proof may lack complete axiomatic details
• Very labor intensive
  – Loop invariants - usually manual
  – Input and output assertions - manual
  – Verification conditions - can be automated
Software Tools Can Help

- Proof Checkers:
  - Scrutinize the steps of a proof and determine if they are sound
  - Identify the rule(s) of inference, axiom(s), etc. needed to justify each step
  - How to know if the proof checker is right (verify it? with what? .....)

Software Tools Can Help

- Verification Assistants
  - Facilitate precise expression of assertions
  - Accept rules of inference
  - Accept axioms
  - Construct statements of needed lemmas
  - Check proofs
  - Assist in construction of proofs (theorem provers)
Human/computer collaboration

- Most successful -- human/computer collaboration
  » Human architects the proof
  » Computer attempts the proof (generally by exhaustive search of space of possible axioms and inferences at each step)
  » Human intervention after computer has tried for a while

Current Status:

- Have verified some non-trivial programs or important parts of programs
  - e.g., protocol verification
  - TOKENEER
- Improved theorem provers
- Improved specification languages
- Verification and testing/analysis research now viewed more as a continuum
  testing--> finite state verification--> verifications
Summary

- Verification has had a very positive impact on software engineering
  - major argument for structured programming
    » Dijkstra's "goto's considered harmful" letter
    » one-in one-out structures easier to reason about
  - major impetus for abstract data types
    » centralized all changes to a data structures
    » input/output assertions for all operations

Formal Development

- Start with assertions, develop software artifacts to fulfill them
- A top-down approach
- Very popular in Europe: A hard sell in the U.S.
- Need to prove lemmas in higher level software dictates the functional requirements (eg. input/output assertion) pairs of lower level software artifacts.
- Also suggests the use of libraries of reusable verified software artifacts for commonly needed utilities
- This is Component-based software development
Integration of Testing Analysis and Formal Methods

- **Testing**
  - Is dynamic in nature, entailing execution of the program
  - Requires skillful selection of test data to assure good exercising of the program
  - Can show program executing in usage environment
  - Can support arbitrarily detailed examination of virtually any program characteristics and behavior
  - Is generally not suitable for showing absence of faults

- **Analysis**
  - Is static, operating on abstract program representations
  - Supports definitive demonstration of absence of faults
  - Generally only for certain selected classes of faults

- **Formal Methods**
  - Most thorough, rigorous, mathematical
  - Apply primarily to checking functional characteristics
  - Most human and cost intensive

- The types of capabilities are complementary; suggests need for skillful integration

---

Definitive reasoning benefits from both static and dynamic analysis techniques

- Religious wars of the 70’s
- Need testing to validate the “ground truth”
- Need static analysis to evaluate more than just what can be examined with testing
- Testing and analysis techniques currently being developed to work together
  - Testing -> Bug -> property -> verification -> counter examples -> feasibility analysis -> test cases -> testing ...
No Need To Restrict this only to Code

- Much of this is applicable to non-code artifacts
- Payoffs from detecting faults is greater the earlier it takes place
- How to apply this to non-code?