Program Verification

Notes by Jonathan Buss
Based in part on materials prepared by B. Bonakdarpour
from Huth & Ryan text,
and material by Anna Lubiw
Additional thanks to D. Maftuleac, R. Trefler, and P. Van Beek
Modified by Lila Kari

Outline

- Introduction: What and Why?
- Pre- and Post-conditions
- Conditionals
- `while`-Loops and Total Correctness

Program Verification

- Reference: Huth & Ryan, Chapter 4
- **Program correctness**: does a given program satisfy its specification—does it do what it is supposed to do?
- Techniques for showing program correctness:
  - inspection, code walk-throughs
  - testing
    - black box: tests designed independent of code
    - while box: tests designed based on code
  - *formal verification*

Program Verification

> *"Testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence."*
>
> [E. Dijkstra, 1972.]

Testing is not proof!

Testing versus Formal Verification

- Testing:
  - check a program for carefully chosen inputs (e.g., boundary conditions, etc.)
  - in general: cannot be exhaustive
- Formal verification:
  - formally state a specification (logic, set theory), and
  - **prove** a program satisfies the specification for **all** inputs
    - although undecidable (= no algorithm) in general, we will study some useful techniques
    - part of *Software Engineering*

Why Formal Verification?

Why formally specify and verify programs?

- Reduce bugs
- Safety-critical software or important components (e.g., brakes in cars, nuclear power plants)
- Documentation
  - necessary for large multi-person, multi-year software projects
  - good documentation facilitates code re-use
- Current Practice
  - specifying software is widespread practice
  - formally verifying software is less widespread
  - hardware verification is commmon

Some Software Bugs

- **Therac-25, X-ray, 1985**
  - overdosing patients during radiation treatment, 5 dead
  - *reason*: race condition between concurrent tasks
- **AT&T, 1990**
  - long distance service fails for 9 hours
  - *reason*: wrong BREAK statement in C code
- **Patriot-Scud, 1991**
  - 28 dead and 100 injured
  - *reason*: rounding error
- **Pentium Processor, 1994**
  - error in division algorithm
  - *reason:* incomplete entries in a look-up table

Some Software Bugs

- **Ariane 5, 1996**
  - exploded 37 seconds after takeoff
  - *reason:* data conversion of a too large number
- **Mars Climate Orbiter, 1999**
  - destroyed on entering atmosphere of Mars
  - *reason:* mixture of pounds and kilograms
- **Power black-out, 2003**
  - 50 million people in Canada and US without power
  - *reason:* programming error
- **Royal Bank, 2004**
  - financial transactions disrupted for 5 days
  - *reason:* programming error

Some Software Bugs

- **UK Child Support Agency, 2004**
    - overpaid 1.9 million people, underpaid 700,000, cost to taxpayers over $1 billion
    - *reason:* more than 500 bugs reported
- **Science (a prestigious scientific journal), 2006**
    - retraction of research papers due to erroneous research results
    - *reason:* program incorrectly flipped the sign (+ to -) on data
- **Toyota Prius, 2007**
    - 160,000 hybrid vehicles recalled due to stalling unexpectedly
    - *reason:* programming error
- **Knight Capital Group, 2012**
    - high-frequency trading system lost $440 million in 30 min
    - *reason:* programming error

Framework for software verification

The steps of formal verification:

1. Convert the informal description R of requirements for an application domain into an “equivalent” formula Φ_R of some symbolic logic,
2. Write a program P which is meant to realise Φ_R in some given programming environment, and
3. **Prove that the program P satisfies the formula Φ_R.**

We shall consider only the third part in this course.

Core programming language

We shall use a subset of C/C++ and Java.
It contains their core features:

- integer and Boolean expressions
- assignment
- sequence
- `if-then-else` (conditional statements)
- `while`-loops
- `for`-loops
- arrays
- functions and procedures

Program States

We are verifying **imperative, sequential, transformational** programs.

- imperative: sequence of commands which modify the values of variables
- sequential: no concurrency
- transformational: given inputs, compute outputs and terminate

Imperative programs

- Imperative programs manipulate variables.
- The **state** of a program is the values of the variables
  at a particular time in the execution of the program.
- Expressions evaluate relative to the current state of the program.
- Commands change the state of the program.

Example

We shall use the following code as an example.

Compute the factorial of input `x` and store in `y`.

```
    y = 1;
    z = 0;
⟶  while (z != x) {
        z = z + 1;
        y = y * z;
    }
```

State at the "`while`" test:

- Initial state s_0: z=0, y=1
- Next state s_1: z=1, y=1
- State s_2: z=2, y=2
- State s_3: z=3, y=6
- State s_4: z=4, y=24

⋮

Note: the order of "`z = z + 1`" and "`y = y * z`" matters!

Example

We shall use the following code as an example.

Compute the factorial of input `x` and store in `y`.

```
    y = 1;
    z = 0;
⟶  while (z != x) {
        z = z + 1;
        y = y * z;
    }
```

State at the "`while`" test:

- Initial state s_0: `z`=0, `y`=1
- Next state s_1: `z`=1, `y`=1
- State s_2: `z`=2, `y`=2
- State s_3: `z`=3, `y`=6
- State s_4: `z`=4, `y`=24

$\vdots$

Note: the order of "`z = z + 1`" and "`y = y * z`" matters!

Specifications

Example.

> Compute a number y whose square is less than the input x.

What if $x = -4$?

Revised example.

> If the input x is a positive number, compute a number whose square is less than x.

For this, we need information not just about the state *after* the program executes, but also about the state *before* it executes.

Hoare Triples

Our assertions about programs will have the form

$(\!| P |\!)$ — **precondition**
C — **program or code**
$(\!| Q |\!)$ — **postcondition**

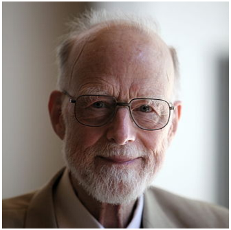
The meaning of the triple $(\!| P |\!)\ C\ (\!| Q |\!)$

> If program C is run starting in a state that satisfies P, then the resulting state after the execution of C will satisfy Q.

An assertion $(\!| P |\!)\ \mathrm{C}\ (\!| Q |\!)$ is called a **Hoare triple**.

Syntax of Hoare Triples

- Conditions P and Q are written in predicate logic of integers
- Use predicates $<, =$, functions $+, -, *$ and others derivable from these
- Tony Hoare (C.A. R. Hoare), b. 1934
- famous for Quicksort and program verification



Specification of a Program

A *specification* of a program C is a Hoare triple with C as the second component: $(\!| P |\!) \ \mathrm{C} \ (\!| Q |\!)$.

Example. The requirement

> If the input x is a positive number, compute a number whose square is less than x

might be expressed as

$$(\!| x > 0 |\!) \ C \ (\!| y \cdot y < x |\!) \ .$$

Specification Is Not Behaviour

Note that a triple $(\!| x > 0 |\!) \ C \ (\!| y * y < x |\!)$ specifies neither a unique program C nor a unique behaviour.

C_1:
```
y = 0 ;
```

C_2:
```
y = 0 ;
while (y * y < x) {
    y = y + 1 ;
    }
y = y - 1 ;
```

Better postcondition

$$(y * y < x) \wedge \forall z((z * z < x) \longrightarrow z \leq y)$$

Hoare triples

We want to develop a notion of proof that will allow us to prove that a program C satisfies the specification given by the precondition P and the postcondition Q.

The proof calculus is different from the proof calculus in first-order (predicate) logic, since it is about proving triples, which are built from two different kinds of things:

- logical formulas: P, Q, and
- code C

Partial correctness

A triple $(\!|\, P \,|\!)\ C\ (\!|\, Q \,|\!)$ is satisfied under partial correctness, denoted

$$\models_{\texttt{par}} (\!|\, P \,|\!)\ C\ (\!|\, Q \,|\!)\ ,$$

if and only if

> for every state s that satisfies condition P,
>
> if execution of C starting from state s terminates in a state s',
>
> then state s' satisfies condition Q.

Partial correctness

In particular, the program

```
while true { x = 0; }
```

satisfies all specifications!

It is an endless loop and never terminates, but partial correctness only says what must happen if the program terminates.

Total correctness

A triple $(\!| P |\!) \; C \; (\!| Q |\!)$ is satisfied under total correctness, denoted

$$\models_{\texttt{tot}} (\!| P |\!) \; C \; (\!| Q |\!) \;,$$

if and only if

> for every state s that satisfies P,
>
> execution of C starting from state s terminates,
>
> and the resulting state s' satisfies Q.

Total Correctness = Partial Correctness + Termination

Examples for Partial and Total Correctness

Example 1.

$(\!|\, x = 1 \,|\!)$

```
y=x;
```

$(\!|\, y = 1 \,|\!)$

Total correctness satisfied.

Example 2.

$(\!|\, x = 1 \,|\!)$

```
y=x  ;
```

$(\!|\, y = 2 \,|\!)$

Neither total nor partial correctness satisfied.

Examples for Partial and Total Correctness

Example 3.

$(\!| \, x = 1 \, |\!)$

```
while (true) {
    x = 0 ;
}
```

$(\!| \, x > 0 \, |\!)$

Infinite loop (partial correctness)

Partial and Total Correctness

Example 4.

$(\!| x \geq 0 |\!)$

```
y = 1 ;
z = 0 ;
while (z != x) {
    z = z + 1 ;
    y = y * z ;
}
```

$(\!| y = x! |\!)$

Total correctness

What happens if we remove pre-condition (replace with “true”)?

Partial correctness but not total correctness: C loops forever on negative input

Examples for Partial and Total Correctness

Example 5.

$(\!|\ x \geq 0\ |\!)$

```
y = 1 ;
while (x != 0) {
    y = y * x ;
    x = x - 1 ;
}
```

$(\!|\ y = x!\ |\!)$

No correctness, because input altered (“consumed”)

Partial correctness is really weak

Give a program that is partially correct for any pre- and post-conditions

```
⦇ P ⦈
while (true){
    x = 0
}
⦇ Q ⦈
```

The program never terminates so partial correctness is vacuously true.

Partial correctness is really weak

At the other extreme, consider

$$\begin{array}{l} (\!|\, true \,|\!) \\ \mathtt{C} \\ (\!|\, true \,|\!) \end{array}$$

Suppose

- C never terminates partial correctness
- C sometimes terminates partial correctness
- C always terminates total correctness

Logical variables

Sometimes in our specifications (pre- and post- conditions) we will need additional variables that do not appear in the program.

These are called logical variables.

Example.

$$(\!|\ x = x_0 \wedge x_0 \geq 0\ |\!)$$

```
y = 1;
while (x != 0) {
    y = y * x;
    x = x - 1;
}
```

$$(\!|\ \mathtt{y} = x_0!\ |\!)$$

For a Hoare triple, its set of logical variables are those variables that are free in P or Q and do not occur in C.

Partial and Total Correctness in Logic

We can write the conditions for partial and total correctness in predicate logic:

- *States*(s) - Predicate: "s is an element of the set of states"
- *Satisfies*(s, P) - Predicate: "State s satisfies condition P"
- *Terminates*(C, s): Predicate: "code C terminates when execution begins in state s"
- *result*(C, s): function: the state that results from executing code C beginning in state s, if C terminates (undefined otherwise)

Partial and Total Correctness in Logic

- Partial correctness of Hoare triple $(\!|\, P \,|\!)\ C\ (\!|\, Q \,|\!)$:

  $\forall s[\mathit{States}(s) \longrightarrow (\mathit{Satisfies}(s, P) \wedge \mathit{Terminates}(C, s) \longrightarrow \mathit{Satisfies}(\mathit{result}(C, s), Q))]$

- Total correctness of Hoare triple

  $\forall s[\mathit{States}(s) \longrightarrow (\mathit{Satisfies}(s, P) \longrightarrow \mathit{Terminates}(C, s) \wedge \mathit{Satisfies}(\mathit{result}(C, s), Q))]$

Proving correctness

- Total correctness is our goal.
- We usually prove it by proving partial correctness and termination separately.
    - For partial correctness, we shall introduce sound inference rules.
    - Proving termination is often easy, but not always (in general, it is undecidable)

Partial and Total Correctness

- Why do we separate into partial/total correctness?
- Both are undecidable, i.e., there is no algorithm to solve them
- There are different techniques for partial and total correctness
- We will look at a proof system for proving partial correctness

Proving Partial Correctness

Recall the definition of Partial Correctness:

> For every starting state which satisfies P and for which C terminates, the final state satisfies Q.

How do we show this, if there are a large or infinite number of possible states?

Answer: **Inference rules** (proof rules, like in formal deduction)

> Rules for each construct in our programming language.

What will a Proof Look Like

An annotated program with conditions before and after every program statement. Each Hoare triple (condition, program statement, condition) will have a justification.

```
⦇ precondition ⦈
y = 1;
⦇ ... ⦈                        ⟨justification⟩
while (x != 0) {
    ⦇ ... ⦈                    ⟨justification⟩
    y = y * x;
    ⦇ ... ⦈                    ⟨justification⟩
    x = x - 1;
    ⦇ ... ⦈                    ⟨justification⟩
}
⦇ postcondition ⦈              ⟨justification⟩
```

Inference Rule for Assignment

$$\frac{}{(\!|\, Q[E/x] \,|\!)\ \ \mathrm{x} = E\ \ (\!|\, Q \,|\!)}\ \text{(assignment)}$$

Intuition:

> $Q(x)$ will hold after assigning (the value of) E to x if $Q(E)$ was true initially.

Note: Normally, Q will be a formula with variable x in it, $Q(x)$

Assignment: Example

Example.

$$\vdash_{\texttt{par}} (\!|\, y + 1 = 7 \,|\!)\ \texttt{x = y + 1}\ (\!|\, x = 7 \,|\!)$$

by one application of the assignment rule.

Examples for Assignment

Example 1.

$$\begin{array}{ll}
(\!|\, y = 2 \,|\!) & (\!|\, P[E/x] \,|\!) \\
\texttt{x = y ;} & x = E; \\
(\!|\, x = 2 \,|\!) & (\!|\, P \,|\!)
\end{array}$$

Here P is "$x = 2$", $E = y$, $P[y/x]$ is "$y = 2$".

Example 2.

$$\begin{array}{ll}
(\!|\, 0 < 2 \,|\!) & (\!|\, P[E/x] \,|\!) \\
\texttt{x = 2 ;} & x = E; \\
(\!|\, 0 < x \,|\!) & (\!|\, P \,|\!)
\end{array}$$

Here P is "$0 < x$", $E = 2$, $P[2/x]$ is "$0 < 2$"

Examples of Assignment

Example 3.

$$\begin{array}{ll} (\!|\, x+1=2 \,|\!) & (\!|\, (x=2)[(x+1)/x] \,|\!) \\ \texttt{x = x + 1;} & x = x+1; \\ (\!|\, x=2 \,|\!) & (\!|\, x=2 \,|\!) \end{array}$$

Here P is "$x = 2$", $E = x + 1$

Example 4.

$$\begin{array}{l} (\!|\, x+1=n+1 \,|\!) \\ \texttt{x = x + 1;} \\ (\!|\, x=n+1 \,|\!) \end{array}$$

Here P is "$x = n + 1$", $E = x + 1$

Note about Examples

In program correctness proofs, we usually work backwards from the postcondition:

??	$\llparenthesis\, P[E/x] \,\rrparenthesis$
`x = y;`	$x = E;$
$\llparenthesis\, x > 0 \,\rrparenthesis$	$\llparenthesis\, P \,\rrparenthesis$

Inference Rules with Implications

Precondition strengthening:

$$\frac{P \rightarrow P' \qquad (\!| P' |\!) \ C \ (\!| Q |\!)}{(\!| P |\!) \ C \ (\!| Q |\!)} \ \textit{(implied)}$$

Postcondition weakening:

$$\frac{(\!| P |\!) \ C \ (\!| Q' |\!) \qquad Q' \rightarrow Q}{(\!| P |\!) \ C \ (\!| Q |\!)} \ \textit{(implied)}$$

Example

$$\frac{P \to P' \qquad (\!|\, P' \,|\!)\ C\ (\!|\, Q \,|\!)}{(\!|\, P \,|\!)\ C\ (\!|\, Q \,|\!)}\ \textit{(implied)}$$

$(\!|\, y = 6 \,|\!)$
$(\!|\, y + 1 = 7 \,|\!)$ implied
`x = y + 1`
$(\!|\, x = 7 \,|\!)$ assignment

Here: P is $y = 6$

P' is $y + 1 = 7$

C is $x = y + 1$

Q is $x = 7$

Note that here $P \longleftrightarrow P'$

Example

$$\frac{(\!|\, P \,|\!)\ \ C\ \ (\!|\, Q' \,|\!) \qquad Q' \to Q}{(\!|\, P \,|\!)\ \ C\ \ (\!|\, Q \,|\!)}\ \textit{(implied)}$$

$(\!|\, y+1=7 \,|\!)$
`x = y+1`
$(\!|\, x=7 \,|\!)$
$(\!|\, x \le 7 \,|\!)$ implied

Here: P is $y+1=7$

C is $x=y+1$

Q' is $x=7$

Q is $x \le 7$.

In this case, $Q' \longrightarrow Q$ but the converse is not true.

Inference Rule for Sequences of Instructions

$$\frac{(\!|\, P \,|\!)\ \ C_1\ \ (\!|\, Q \,|\!),\quad (\!|\, Q \,|\!)\ \ C_2\ \ (\!|\, R \,|\!)}{(\!|\, P \,|\!)\ \ C_1;\ C_2\ \ (\!|\, R \,|\!)}\ \textit{(composition)}$$

In order to prove $(\!|\, P \,|\!)\ \ C_1;\ \ C_2\ \ (\!|\, R \,|\!)$, we need to find a midcondition Q for which we can prove $(\!|\, P \,|\!)\ \ C_1\ \ (\!|\, Q \,|\!)$ and $(\!|\, Q \,|\!)\ \ C_2\ \ (\!|\, R \,|\!)$.

(In our examples, the mid-condition will usually be determined by a rule, such as assignment. But in general, a mid-condition might be very difficult to determine.)

- Inference rules with sequence of instructions allow us to string together pre/postconditions and lines of code
- Each condition is the postcondition of the previous line of code and the precondition of the next line of code

Proof Format: Annotated Programs

- Interleave program statements with assertions (= conditions), each justified by an inference rule.
- The composition rule is implicit.
- Each assertion should hold whenever the program reaches that point in its execution.
- Each assertion is justified by an inference rule
  - If implied inference rule is used, we also need to prove the implication. This is done after annotating the program.
  - don't simplify assertions in the annotated program. Do them as implied inferences.

Example: Composition of Assignments

To show: the following is satisfied under partial correctness.

We work bottom-up for assignments. . .

$$
\begin{array}{ll}
(\!|\, x = x_0 \land y = y_0 \,|\!) & \\
(\!|\, y = y_0 \land x = x_0 \,|\!) & (\!|\, P_3[x/t] \,|\!) \\
\texttt{t = x ;} & \\
(\!|\, y = y_0 \land t = x_0 \,|\!) & P_3 = (\!|\, P_2[y/x] \,|\!) \\
\texttt{x = y ;} & \\
(\!|\, x = y_0 \land t = x_0 \,|\!) & P_2 = (\!|\, P[t/y] \,|\!) \\
\texttt{y = t ;} & \\
(\!|\, x = y_0 \land y = x_0 \,|\!) & \text{assignment} \quad (\!|\, P \,|\!)
\end{array}
$$

Finally, show $(\!|\, x = x_0 \land y = y_0 \,|\!)$ implies $(\!|\, y = y_0 \land x = x_0 \,|\!)$.

Example 1 and Comments

$(\!| y = 5 |\!)$

$(\!| y + 1 = 6 |\!)$ implied

`x = y+1;`

$(\!| x = 6 |\!)$ assignment

- The proof is constructed from the bottom upwards
- We start with $x = 6$ and, using the assignment rule, we push it upwards through (the assignment) $x = y + 1$
- This means substituting $y + 1$ for all occurrences of x, resulting in $y + 1 = 6$
- Now compare this with the given precondition $y = 5$.
- The given precondition and the arithmetic fact that 5+1 =6 imply it, so we have finished the proof

Example 1 and Comments

- Although the proof is constructed bottom-up, its justifications make sense when read top-down
- The 2nd line is implied by the 1st line
- The 4th line follows from the 2nd, by the intervening assignment $x = y + 1$
- Note that implied always refers to the immediately preceding line
- Proofs in program logic generally combine two logical levels
  - The 1st is directly concerned with proof rules for programming constructs, such as the assignment statement
  - The 2nd level is ordinary logic derivations (as familiar from propositional and predicate logic) plus facts from arithmetic.

Example 2 and Comments

$(\!|\, y < 3 \,|\!)$
$(\!|\, y + 1 < 4 \,|\!)$ implied
```
y = y+1;
```
$(\!|\, y < 4 \,|\!)$ assignment

- We may use ordinary logical and arithmetic implications to change a certain condition φ to any condition φ' which is implied by φ (that is, $\varphi \longrightarrow \varphi'$) for reasons which have nothing to do with the code
- Here, φ was $y < 3$ and the implied formula φ' was $y + 1 < 4$.
- The validity of this implication is rooted in general facts about integers and the relation $<$.
- Completely formal proofs would require separate proofs attached to all instances of the rule implied.
- We will not always do that.

Programs with Conditional Statements

Deduction Rules for Conditionals

`if-then-else`:

$$\frac{(\!|\, P \wedge B \,|\!)\ \mathtt{C_1}\ (\!|\, Q \,|\!) \qquad (\!|\, P \wedge \neg B \,|\!)\ \mathtt{C_2}\ (\!|\, Q \,|\!)}{(\!|\, P \,|\!)\ \mathtt{if\ (B)\ C_1\ else\ C_2}\ (\!|\, Q \,|\!)}\ \text{(if-then-else)}$$

`if-then` (without `else`):

$$\frac{(\!|\, P \wedge B \,|\!)\ \mathtt{C}\ (\!|\, Q \,|\!) \qquad (P \wedge \neg B) \to Q}{(\!|\, P \,|\!)\ \mathtt{if\ (B)\ C}\ (\!|\, Q \,|\!)}\ \text{(if-then)}$$

Template for Conditionals With `else`

Annotated program template for `if-then-else`:

```
$\llparenthesis P \rrparenthesis$
if ( $B$ ) {
    $\llparenthesis P \land B \rrparenthesis$        if-then-else
    $C_1$
    $\llparenthesis Q \rrparenthesis$            (justify depending on $C_1$—a “subproof”)
} else {
    $\llparenthesis P \land \neg B \rrparenthesis$       if-then-else
    $C_2$
    $\llparenthesis Q \rrparenthesis$            (justify depending on $C_2$—a “subproof”)
}
$\llparenthesis Q \rrparenthesis$                if-then-else [justifies this $Q$, given previous two]
```

Template for Conditionals Without `else`

Annotated program template for `if-then`:

```
$(\!| P |\!)$
if ( $B$ ) {
    $(\!| P \wedge B |\!)$        if-then
    $C$
    $(\!| Q |\!)$                 [add justification based on $C$]
}
$(\!| Q |\!)$                     if-then
                                  Implied: Proof of $P \wedge \neg B \to Q$
```

Example: Conditional Code

Example: Prove the following is satisfied under partial correctness.

$(\!|\, true \,|\!)$

```
if ( max < x ) {
    max = x ;
}
```

$(\!|\, max \geq x \,|\!)$

$(\!|\, P \,|\!)$

```
if ( B ) {
    C
}
```

$(\!|\, Q \,|\!)$

First, let's recall our proof method. . . .

The Steps of Creating a Proof

Three steps in doing a proof of partial correctness:

1. First annotate using the appropriate inference rules.
2. Then ''back up'' in the proof: add an assertion before each assignment statement, based on the assertion following the assignment.
3. Finally prove any "implieds":
    - Annotations from (1) above containing implications
    - Adjacent assertions created in step (2).

   Proofs here can use predicate logic, basic arithmetic, or other appropriate reasoning.

Doing the Steps

1. Add annotations for the `if-then` statement.
2. "Push up" for the assignments.
3. Identify "implieds" to be proven.

$(\!|\ \mathrm{true}\ |\!)$

```
if ( max < x ) {
```

$(\!|\ true \wedge max < x\ |\!)$ if-then

$(\!|\ x \geq x\ |\!)$ Implied (a)

```
    max = x ;
```

$(\!|\ max \geq x\ |\!)$ $\longleftarrow$ *to be shown*

```
}
```

$(\!|\ max \geq x\ |\!)$ if-then

Implied: $\big(\mathrm{true} \wedge \neg(max < x)\big) \rightarrow max \geq x$

Proving "Implied" Conditions

The auxiliary "implied" proofs can be done by Natural Deduction (and assuming the necessary arithmetic properties). We will use it informally.

Proof of Implied (a):

$$\vdash \Big((\mathit{true} \wedge (\mathit{max} < x)\Big) \rightarrow x \geq x \ \ .$$

Clearly $x \geq x$ is a tautology and the implication holds.

Implied (b)

Proof of Implied (b): Show $\vdash (P \wedge \neg B) \rightarrow Q$, which is

$$\emptyset \vdash \big(true \wedge \neg(max < x)\big) \rightarrow (max \geq x) \ .$$

1. $\big(true \wedge \neg(max < x)\big) \vdash \big(true \wedge \neg(max < x)\big)$ $(\in)$
2. $\big(true \wedge \neg(max < x)\big) \vdash \neg(max < x)\big)$ $(1, \wedge -)$
3. $\big(true \wedge \neg(max < x)\big) \vdash (max \geq x)$ $(def.of \geq)$
4. $\emptyset \vdash \big(true \wedge \neg(max < x)\big) \rightarrow (max \geq x)$

Example 2 for Conditionals

Prove the following is satisfied under partial correctness.

$(\!|\ \text{true}\ |\!)$

```
if (x > y) {
    max = x;
} else {
    max = y;
}
```

$(\!|\ (x > y \land max = x) \lor (x \leq y \land max = y)\ |\!)$

Example 2: Annotated Code

```
$(\!|\ \mathrm{true}\ |\!)$
if (x > y) {
    $(\!|\ x > y\ |\!)$                                                          if-then-else
    $(\!|\ (x > y \land x = x) \lor (x \leq y \land x = y)\ |\!)$                implied (a)
    max = x ;
    $(\!|\ (x > y \land max = x) \lor (x \leq y \land max = y)\ |\!)$            assignment
} else {
    $(\!|\ \neg(x > y)\ |\!)$                                                    if-then-else
    $(\!|\ (x > y \land y = x) \lor (x \leq y \land y = y)\ |\!)$                implied (b)
    max = y ;
    $(\!|\ (x > y \land max = x) \lor (x \leq y \land max = y)\ |\!)$            assignment
}
$(\!|\ (x > y \land max = x) \lor (x \leq y \land max = y)\ |\!)$                if-then-else
```

Example 2: Implied Conditions

(a) Prove $\emptyset \vdash x > y \to (x > y \land x = x) \lor (x \leq y \land x = y)$

1. $x > y \vdash x > y$ $(\in)$
2. $\emptyset \vdash x = x$ $(\approx +)$
3. $x > y \vdash x = x$ $(2, +)$
4. $x > y \vdash x > y \land x = x$ $(1, 3, \land +)$
5. $x > y \vdash (x > y \land x = x) \lor (x \leq y \land x = y)$ $(4, \lor +)$
6. $\emptyset \vdash x > y \to (x > y \land x = x) \lor (x \leq y \land x = y)$ $(4, \to +)$

Example 2 for Conditionals

(b) Prove $x \leq y \rightarrow \big((x > y \wedge x = x) \vee (x \leq y \wedge y = y)\big)$.

1. $x \leq y \vdash x \leq y$ $(\in)$
2. $\emptyset \vdash y = y$ $(\approx +)$
3. $x \leq y \vdash y = y$ $(2, +)$
4. $x \leq y \vdash x \leq y \wedge y = y$ $(1, 3, \wedge +)$
5. $x \leq y \vdash (x > y \wedge x = x) \vee (x \leq y \wedge y = y)$ $(4, \vee +)$
6. $\emptyset \vdash x \leq y \rightarrow (x > y \wedge x = x) \vee (x \leq y \wedge y = y)$ $(5, \rightarrow +)$

`While`-Loops and Total Correctness

Inference Rule: Partial-`while`

"Partial while": do not (yet) require termination.

$$\frac{(\!|\, I \wedge B \,|\!)\ \ C\ \ (\!|\, I \,|\!)}{(\!|\, I \,|\!)\ \texttt{while}\ (B)\ C\ \ (\!|\, I \wedge \neg B \,|\!)}\ (\text{partial-}\texttt{while})$$

In words:

> If the code C satisfies the triple $(\!|\, I \wedge B \,|\!)\ \ C\ \ (\!|\, I \,|\!)$,
> and I is true at the start of the `while`-loop,
> then no matter how many times we execute C,
> condition I will still be true.

Condition I is called a *loop invariant*.

After the `while`-loop terminates, $\neg B$ is also true.

Annotations for Partial-`while`

```
$\llparenthesis P \rrparenthesis$
$\llparenthesis I \rrparenthesis$                      Implied (a)
while ( $B$ ) {
    $\llparenthesis I \land B \rrparenthesis$              partial-while
    $C$
    $\llparenthesis I \rrparenthesis$                  $\longleftarrow$ to be justified, based on $C$
    }
$\llparenthesis I \land \neg B \rrparenthesis$             partial-while
$\llparenthesis Q \rrparenthesis$                      Implied (b)
```

(a) Prove $P \to I$ (precondition P implies the loop invariant)

(b) Prove $(I \land \neg B) \to Q$ (exit condition implies postcondition)

We need to determine I!!

Loop Invariants

A *loop invariant* is an assertion (condition) that is true both *before* and *after* each execution of the body of a loop.

- True before the `while`-loop begins.
- True after the `while`-loop ends.
- Expresses a relationship among the variables used within the body of the loop. Some of these variables will have their values changed within the loop.
- An invariant may or may not be useful in proving termination (to discuss later).

Example: Finding a loop invariant

$(\!| x \geq 0 |\!)$

```
   y = 1 ;
   z = 0 ;
⟶ while (z != x) {
       z = z + 1 ;
       y = y * z ;
   }
```

$(\!| y = x! |\!)$

At the `while` statement:

x	y	z	$z \neq x$
5	1	0	true
5	1	1	true
5	2	2	true
5	6	3	true
5	24	4	true
5	120	5	false

From the trace and the post-condition,
a candidate loop invariant is $y = z!$

Why are $y \geq z$ or $x \geq 0$ not useful?

These do not involve the loop-termination condition.

Annotations Inside a `while`-Loop

1. First annotate code using the while-loop inference rule, and any other control rules, such as `if-then`.
2. Then work bottom-up (“push up”) through program code.
    - Apply inference rule appropriate for the specific line of code, or
    - Note a new assertion (“implied”) to be proven separately.
3. Prove the implied assertions using the inference rules of ordinary logic.

Example: annotations for partial-`while`

Annotate by partial-`while`, with chosen invariant $(y = z!)$. Annotate assignment statements (bottom-up). Note the required implied conditions.

```
$\llparenthesis x \geq 0 \rrparenthesis$
$\llparenthesis 1 = 0! \rrparenthesis$                            implied (a)
y = 1 ;
$\llparenthesis y = 0! \rrparenthesis$                            assignment
z = 0 ;
$\llparenthesis y = z! \rrparenthesis$                            assignment
while (z != x) {
    $\llparenthesis (y = z!) \wedge \neg(z = x) \rrparenthesis$           partial-while ($\llparenthesis I \wedge B \rrparenthesis$)
    $\llparenthesis y(z+1) = (z+1)! \rrparenthesis$               implied (b)
    z = z + 1 ;
    $\llparenthesis yz = z! \rrparenthesis$                       assignment
    y = y * z ;
    $\llparenthesis y = z! \rrparenthesis$                        assignment
}
$\llparenthesis y = z! \wedge z = x) \rrparenthesis$                  partial-while ($\llparenthesis I \wedge \neg B \rrparenthesis$)
$\llparenthesis y = x! \rrparenthesis$                            implied (c)
```

Example: implied conditions (a) and (c)

Proof of implied (a): $(x \geq 0) \vdash (1 = 0!)$.

By definition of factorial.

Proof of implied (c): $\big((y = z!) \wedge (z = x)\big) \vdash (y = x!)$.

1. $(y = z!) \wedge (z = x) \vdash (y = z!) \wedge (z = x)$ $(\in)$
2. $(y = z!) \wedge (z = x) \vdash (y = z!)$ $(1, \wedge -)$
3. $(y = z!) \wedge (z = x) \vdash (z = x)$ $(1, \wedge -)$
4. $(y = z!) \wedge (z = x) \vdash (y = x!)$ $(2, 3, \approx -)$

Example: implied condition (b)

Proof of implied (b):

$$\big((y = z!) \land \neg(z = x)\big) \vdash (z+1)\,y = (z+1)! \ \ .$$

1. $y = z! \land z \neq x \vdash y = z! \land z \neq x \quad (\in)$
2. $y = z! \land z \neq x \vdash y = z! \quad (1, \land-))$
3. $(z+1)y = (z+1)z! \quad (2, algebra)$
4. $(z+1)z! = (z+1)! \quad (def. \ of \ factorial)$
5. $(z+1)y = (z+1)! \quad (3, 4, transitivity \ of \ equality)$

Example 2 (Partial-`while`)

Prove the following is satisfied under partial correctness.

$(\!|\, n \geq 0 \wedge a \geq 0 \,|\!)$

```
s = 1 ;
i = 0 ;
while (i < n) {
    s = s * a ;
    i = i + 1 ;
}
```

$(\!|\, s = a^n \,|\!)$

Trace of the loop:

a	n	i	s
2	3	0	1
2	3	1	$1*2$
2	3	2	$1*2*2$
2	3	3	$1*2*2*2$

Candidate for the loop invariant: $s = a^i$.

Example 2: Testing the invariant

Using $s = a^i$ as an invariant yields the annotations shown at right.

Next, we want to

- Push up for assignments
- Prove the implications

But: implied (c) is false!

We must use a different invariant.

```
(| n ≥ 0 ∧ a ≥ 0 |)
(| ... |)
s = 1 ;
(| ... |)
i = 0 ;
(| s = a^i |)
while (i < n) {
  (| s = a^i ∧ i < n |)        partial-while
  (| ... |)
  s = s * a ;
  (| ... |)
  i = i + 1 ;
  (| s = a^i |)
}
(| s = a^i ∧ i ≥ n |)          partial-while
(| s = a^n |)                  implied (c)
```

Example 2: Adjusted invariant

Try a new invariant:

$$s = a^i \wedge i \leq n \ .$$

Now the “implied” conditions are actually true, and the proof can succeed.

```
$(\!| n \geq 0 \wedge a \geq 0 |\!)$
$(\!| 1 = a^0 \wedge 0 \leq n |\!)$                      implied (a)
s = 1 ;
$(\!| s = a^0 \wedge 0 \leq n |\!)$                      assignment
i = 0 ;
$(\!| s = a^i \wedge i \leq n |\!)$                      assignment
while (i < n) {
    $(\!| s = a^i \wedge i \leq n \wedge i < n |\!)$     partial-while
    $(\!| s \cdot a = a^{i+1} \wedge i+1 \leq n |\!)$    implied (b)
    s = s * a ;
    $(\!| s = a^{i+1} \wedge i+1 \leq n |\!)$            assignment
    i = i + 1 ;
    $(\!| s = a^i \wedge i \leq n |\!)$                  assignment
}
$(\!| s = a^i \wedge i \leq n \wedge i \geq n |\!)$      partial-while
$(\!| s = a^n |\!)$                                      implied (c)
```

Total Correctness (Termination)

Total Correctness = Partial Correctness + Termination

Only `while`-loops can be responsible for non-termination in our programming language.

(In general, recursion can also cause it).

Proving termination:
For each `while`-loop in the program,

> Identify an integer expression which is *always* non-negative and whose value *decreases* every time through the while-loop.

Example For Total Correctness

The code below has a “*loop guard*” of $z \neq x$,
which is equivalent to $x - z \neq 0$.

What happens to the value of $x - z$ during execution?

```
(| x ≥ 0 |)
y = 1 ;
z = 0 ;

while ( z != x ) {
    z = z + 1 ;
    y = y * z ;
}
(| y = x! |)
```

At start of loop: $x - z = x \geq 0$

`z = z + 1 ;` $x - z$ decreases by 1

`y = y * z ;` $x - z$ unchanged

Thus the value of $x - z$ will eventually reach 0.
The loop then exits and the program terminates.

Proof of Total Correctness

We chose an expression $x - z$ (called the *variant*).

At the start of the loop, $x - z \geq 0$:

- Precondition: $x \geq 0$.
- Assignment $z \leftarrow 0$.

Each time through the loop:

- x doesn't change: no assignment to it.
- z increases by 1, by assignment.
- Thus $x - z$ decreases by 1.

Thus the value of $x - z$ will eventually reach 0.

When $x - z = 0$, the guard $z\, != x$ ends the loop.

Total Correctness Problem

Total Correctness Problem: Given a Hoare triple $(\!| P |\!) \;\; C \;\; (\!| Q |\!)$ is it totally correct?

Theorem The Total Correctness Problem is undecidable.

Proof:

- Reduce the Blank-Tape Halting Problem to our problem.
- Suppose we have an algorithm A to solve the Total Correctness Problem.
- We can use it to solve the Blank-Tape Halting Problem.
- Given program C as input, we can use our algorithm A to test if $(\!| \mathit{true} |\!) \;\; C \;\; (\!| \mathit{true} |\!)$ is totally correct.
- Claim: The program C halts on a blank tape iff this Hoare triple is totally correct.
- Contradiction since the Blank-Tape Halting Problem is undecidable.

Partial Correctness Problem

Partial Correctness Problem: Given a Hoare triple $(\!| P |\!) \;\; C \;\; (\!| Q |\!)$ is it partially correct?

Theorem The Partial Correctness Problem is undecidable.

Proof:

- Reduce the Blank-Tape Halting Problem to our problem.
- Suppose we have an algorithm A to solve the Partial Correctness Problem. We can use it to solve the Blank-Tape Halting Problem for any program C as follows.
- Given program C as input, make a new program C' by adding a new line at the end of the program C (here x is a new variable):

$$x = 1;$$

- Claim: The Hoare Triple $(\!| \mathit{true} |\!) \; C' \; (\!| \mathrm{x}{=}0 |\!)$ is partially correct iff C' does not halt.
- Contradiction since the Blank-Tape Halting Problem is undecidable.

Comments

Where did our method for proving partial/total correctness fail to be an algorithm?

- finding an invariant for while loops
- finding a variant to prove that while loops terminate
- proving the implied conditions - recall that validity in first order (predicate) logic is undecidable.

Logic and Computation: Summary

Propositional Logic

- Translations from English to propositional logic formulas
- Syntax - well formed formulas, structural induction
- Semantics (truth tables, value assignments)
- Proving validity of arguments expressed in propositional logic (by truth tables or by contradiction)
- Propositional calculus laws and normal forms (CNF, DNF)
- Adequate sets of connectives
- Applications of propositional logic: Logic gates, circuits, code simplification
- Formal (natural) deduction, 11 rules, its soundness and completeness
- Resolution
- Davis Putnam Procedure, its soundness and completeness
- Solving the Satisfiability problem with DNA computing

Predicate logic (first-order logic)

- Translations from English to predicate logic formulas
- Syntax - well-formed formulas in predicate logic
- Semantics - interpretations, domains, satisfiability, validity
- Proving validity of arguments expressed in predicate logic
- Formal deduction for predicate logic (17 rules)
- Resolution theorem proving
- Soundness and completeness of formal deduction for predicate logic (Godel)

Undecidability, Applications and Implications

- Undecidability, Halting Problem, other undecidable problems
- Applications and implications of predicate logic
  - Peano Arithmetic
  - Godel's Incompleteness Theorem
  - Program Verification
- Solve logical puzzles and debug invalid arguments

What's wrong with this argument?

Use Logic Wisely!

- THE END -