Program Verification
Outline

• Introduction: What and Why?
• Pre- and Postconditions
• Conditionals
• while-Loops and Total Correctness
• Arrays
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 (white box, black box)
- *formal verification*
"Testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence, never their absence."

[E. Dijkstra, 1972.]
"Testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence, never their absence."

[E. Dijkstra, 1972.]

Testing is not proof!
• 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
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 common
The steps of formal verification:

1. Convert the informal description $R$ of requirements for an application domain into an “equivalent” formula $\Phi_R$ of some symbolic logic,

2. Write a program $P$ which is meant to realise $\Phi_R$ in some given programming environment, and

3. Prove that the program $P$ satisfies the formula $\Phi_R$.

We shall consider only the third part in this course.
Broadly speaking, there are two kinds of programming languages in wide use: procedural and functional.

- *Procedural*, or *imperative*, languages describe procedures to compute something. They specify instructions (imperatives) to follow, in order to achieve the desired result.

- *Functional*, or *applicative*, languages define the results of functions. They specify which functions to apply, in order to obtain the desired result.
Example Programs

As examples, we will use the “powering” function — given $x$ and $y$, compute $x^y$ — on unsigned integers.

Procedural (C++)

```c++
unsigned power ( unsigned x, unsigned y ) {
  if ( y == 0 )
    return 1 ;
  else
    return power( x, y-1 ) * x ;
}
```

Functional (Haskell)

```haskell
power :: Word -> Word
    -> Word
power x y
  | y == 0 = 1
  | otherwise = power x (y-1) * x
```

Different syntax, but basically the same computation.
Exponentiation in Peano Arithmetic

Possible axioms for Peano Arithmetic (symbol ‘↑’):

Exp1: \( \forall x \ (x \uparrow 0 = s(0)) \).

Exp2: \( \forall x \ (\forall y \ (x \uparrow s(y) = (x \uparrow y) \times x)) \).

Expressed in more algorithmic language, these give

\[
x \uparrow y = \begin{cases} 
1 & \text{if } y = 0 \\
(x \uparrow (y - 1)) \times x & \text{otherwise}
\end{cases}
\]

(By convention, the free variables are implicitly universally quantified.)

This makes a third representation of the algorithm in the two programs.
Handling of Variables

Two different versions of power:

With call by value:

```c
unsigned
power ( unsigned x,
        unsigned y ) {
    if ( y == 0 )
        return 1 ;
    else
        return
            power( x, y-1 ) * x ;
}
```

With call by reference:

```c
unsigned
power ( unsigned & x,
        unsigned & y ) {
    if ( y == 0 )
        return 1 ;
    else
        return
            power( x, y-1 ) * x ;
}
```

The call-by-reference version doesn’t compile, since “y-1” isn’t a reference.
A Re-write

Change “y-1” to “y--”:

With call by value:

```c
unsigned
power ( unsigned x,
    unsigned y ) {
    if ( y == 0 )
        return 1 ;
    else
        return
            power( x, y-- ) * x ;
}
```

With call by reference:

```c
unsigned
power ( unsigned & x,
    unsigned & y ) {
    if ( y == 0 )
        return 1 ;
    else
        return
            power( x, y-- ) * x ;
}
```

Now the call-by-reference version compiles — but it’s broken!

Calling “power( a, b )” by reference changes the value of b: a side effect.
An Iterative Implementation

The same effect occurs in iterative versions of the program:

```c
unsigned power ( unsigned x, unsigned y ) {
    unsigned result = 1;
    for ( ; y > 0 ; y-- )
        result *= x;
    return result;
}
```

```c
unsigned power ( unsigned & x, unsigned & y ) {
    unsigned result = 1;
    unsigned z = y;
    for ( ; z > 0 ; z-- )
        result *= x;
    return result;
}
```

In procedural code, the values of variables change!
We shall use a subset of C[++]\, Java, etc.. It contains their core features:

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

Due to time constraints, we shall not include functions and procedures.
We are considering **imperative** or **procedural** programs.

- The programs manipulate the values of “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.
- Executing a statement changes the state of the program.
Example

We shall use the following code as an example.

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

```c
y = 1;
z = 0;

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

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

\[ y = 1; \]
\[ z = 0; \]
\[ \rightarrow \text{while} \ (z \neq x) \ {\}
\]
\[ \quad z = z + 1; \]
\[ \quad y = y \times z; \]
\[ \}

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 “\( y = y \times z \)” matters!
Example

\begin{verbatim}
y = 1;
z = 0;
\rightarrow while (z != x) {
    z = z + 1;
    y = y * z;
}\end{verbatim}

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!
Specifications

What does a “specification” specify?

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.
Specifications

What does a “specification” specify?

Example.

Compute a number \( y \) whose square is less than the input \( x \).

What if \( x = -4 \)?
Specifications

What does a “specification” specify?

**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 $x$. 
What does a “specification” specify?

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

\[ \langle P \rangle \quad C \quad \langle Q \rangle \]

- \( P \) — precondition
- \( C \) — program or code
- \( Q \) — postcondition

The meaning of the triple \( \langle P \rangle \ C \langle Q \rangle \):

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 \( \langle P \rangle \ C \langle Q \rangle \) is called a Hoare triple.
A specification of a program $C$ is a Hoare triple with $C$ as the second component: $⦇ P ⦈ C ⦇ Q ⦈$.

**Example.** The requirement

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

might be expressed as

$$⦇ x > 0 ⦈ C ⦇ y \times y < x ⦈$$.
Specification Is Not Behaviour

A triple, such as \( \langle x > 0 \rangle \ C \ \langle y \times y < x \rangle \), specifies neither a unique program C nor a unique behaviour.

For example, both \( \langle x > 0 \rangle \ C_1 \ \langle y \times y < x \rangle \) and \( \langle x > 0 \rangle \ C_2 \ \langle y \times y < x \rangle \) hold:

\[ \begin{align*}
C_1: & \quad y = 0 ; \\
C_2: & \quad y = 0 ; \\
& \quad \text{while} \ (y \times y < x) \ \{ \\
& \quad \quad y = y + 1 ; \\
& \quad \} \\
& \quad y = y - 1 ;
\end{align*} \]
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 that for FOL, since Hoare triples have two features not present in logical formulas:

- program instructions (actions), rather than propositions, and
- a sense of time: before execution versus after execution.
Partial correctness

A triple $⟨P⟩_C⟨Q⟩$ is satisfied under partial correctness, denoted

$$\models_{\text{par}} ⟨P⟩_C⟨Q⟩,$$

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 $Q$. 

Program Verification Program States and Correctness
Partial correctness

For example, the program

```c
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.
A triple \( (P \downarrow C \downarrow Q) \) is satisfied under total correctness, denoted

\[ \models_{\text{tot}} (P \downarrow C \downarrow Q), \]

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. Total correctness satisfied:

\[
\begin{align*}
\langle x &= 1 \rangle \\
y &= x ; \\
\langle y &= 1 \rangle 
\end{align*}
\]

Example 2. Neither total nor partial correctness:

\[
\begin{align*}
\langle x &= 1 \rangle \\
y &= x ; \\
\langle y &= 2 \rangle 
\end{align*}
\]
Example 3. Infinite loop (partial correctness)

\( \{ x = 1 \} \)

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

\( \{ x > 0 \} \)
Example 4. Total correctness

\[(x \geq 0)\]
\[y = 1;\]
\[z = 0;\]
\[\text{while } (z \neq x) \{\]
\[\quad z = z + 1;\]
\[\quad y = y \times z;\]
\[\}\]
\[(y = x!)]

What happens if we remove the precondition?
Example 5. No correctness, because input altered (“consumed”)

\[
\begin{align*}
\{ & x \geq 0 \} \\
\text{y} = 1 & ; \\
\text{while (x \neq 0) } \{ \\
\text{y} = y \times x & ; \\
\text{x} = x - 1 & ; \\
\} \\
\{ \text{y} = x! & \}
\end{align*}
\]
Logical variables

Sometimes the pre- and postconditions require additional variables that do not appear in the program.

These are called **logical variables**.

**Example.**

\[
\begin{align*}
\langle & x = x_0 \land x_0 \geq 0 \rangle \\
y = 1; \\
\text{while} \ (x \neq 0) \ {\{} \\
& y = y \times x; \\
& x = x - 1; \\
\} \\
\langle & y = x_0! \rangle
\end{align*}
\]
Logical variables

Sometimes the pre- and postconditions require additional variables that do not appear in the program.

These are called **logical variables**.

**Example.**

```
⦇ x = x₀ ∧ x₀ ≥ 0 ⦈

y = 1;
while (x != 0) {
    y = y * x;
    x = x - 1;
}
⦇ y = x₀! ⦈
```

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

• 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.
  • For total correctness, we shall use *ad hoc* reasoning, which suffices for our examples.
    (In general, total correctness is undecidable.)

Our focus on partial correctness may seem strange. It’s not the condition we want to justify.

But experience has shown it is useful to think about partial correctness separately from termination.
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)
Each construct in our programming language will have a rule.
Presentation of a Proof

A full proof will have one or more conditions before and after each code statement. Each statement makes a Hoare triple with the preceding and following conditions. Each triple (postcondition) has a justification that explains its correctness.

\[
\begin{align*}
\{ \text{program precondition} \} \\
y &= 1; \\
\{ \ldots \} \quad \langle \text{justification} \rangle \\
\text{while } (x != 0) \{ \\
\{ \ldots \} \quad \langle \text{justification} \rangle \\
y &= y \times x; \\
\{ \ldots \} \quad \langle \text{justification} \rangle \\
x &= x - 1; \\
\{ \ldots \} \quad \langle \text{justification} \rangle \\
\}
\end{align*}
\]

\[
\{ \text{program postcondition} \} \quad \langle \text{justification} \rangle
\]
Inference Rule for Assignment

\[
( Q[E/x] \land x = E ) \vdash Q
\]

(assignment)

Intuition:

\( Q(x) \) will hold after assigning (the value of) \( E \) to \( x \) if \( Q \) was true of that value beforehand.
Example.

\[ \vdash_{par} (\ y + 1 = 7 \) \ x = y + 1 \ (\ x = 7 \) \]

by one application of the assignment rule.
Example 1.

\[
\begin{align*}
&\neg y = 2 ; \\
& x = y ; \\
& \neg x = 2 ; \\
& (Q[E/x]) \\
& (Q)
\end{align*}
\]

Example 2.

\[
\begin{align*}
&\neg 0 < 2 ; \\
& x = 2 ; \\
& \neg 0 < x ; \\
& (Q[E/x]) \\
& (Q)
\end{align*}
\]
Examples of Assignment

Example 3.

\[ x + 1 = 2 \]
\[ (x = 2) \left[ \frac{(x + 1)}{x} \right] \]
\[ x = x + 1; \]
\[ x = 2 \]

Example 4.

\[ x + 1 = n + 1 \]
\[ (\text{equivalent to } x = n) \]
\[ x = x + 1; \]
\[ x = n + 1 \]
Note about Examples

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

$$??? \quad (\{ Q[E/x] \}$$

$$x = y; \quad x = E;$$

$$\{ x > 0 \} \quad \{ Q \}$$
Inference Rules with Implications

Rule of “Precondition strengthening”:

\[
\begin{align*}
P & \rightarrow P' \\
\{ P' \} & C \{ Q \} \\
\{ P \} & C \{ Q \} \quad \text{(implied)}
\end{align*}
\]

Rule of “Postcondition weakening”:

\[
\begin{align*}
\{ P \} & C \{ Q' \} \\
Q' & \rightarrow Q \\
\{ P \} & C \{ Q \} \quad \text{(implied)}
\end{align*}
\]

Example of use:

\[
\begin{align*}
\{ y = 6 \} \\
\{ y + 1 = 7 \} & \quad \text{implied} \\
x & = y + 1 \\
\{ x = 7 \} & \quad \text{assignment}
\end{align*}
\]
In order to prove $\langle P \rangle \ C_1 \ \langle Q \rangle \ C_2 \ \langle R \rangle$, we need to find a midcondition $Q$ for which we can prove $\langle P \rangle \ C_1 \ \langle Q \rangle$ and $\langle Q \rangle \ C_2 \ \langle R \rangle$.

(In our examples, the midcondition will usually be determined by a rule, such as assignment. But in general, a midcondition might be very difficult to determine.)
Interleave program statements with **assertions**, each justified by an inference rule.

The composition rule is implicit.

Assertions should hold true whenever the program reaches that point in its execution.
Proof Format: Annotated Programs

If implied inference rule is used, we must supply a proof of the implication.

• We’ll do these proofs after annotating the program.

Each assertion should be an instance of an inference rule. Normally,

• Don’t simplify the assertions in the annotated program.
• Do the simplification while proving the implied conditions.
Example: Composition of Assignments

To show: the following is satisfied under partial correctness.

We work bottom-up for assignments...

\[
\{ x = x_0 \land y = y_0 \}
\]

\[
t = x ;
\]

\[
x = y ;
\]

\[
y = t ;
\]

\[
\{ x = y_0 \land y = x_0 \}
\]
Example: Composition of Assignments

To show: the following is satisfied under partial correctness.

We work bottom-up for assignments...

\[
\begin{align*}
\{ x = x_0 \land y = y_0 \} \\
\text{t} = \text{x} ; \\
\text{x} = \text{y} ; \\
\{ x = y_0 \land \text{t} = x_0 \} & \quad P_2 \text{ is } \{ P[t/y] \} \\
\text{y} = \text{t} ; \\
\{ x = y_0 \land y = x_0 \} & \quad \text{assignment } \{ P \}
\end{align*}
\]
Example: Composition of Assignments

To show: the following is satisfied under partial correctness.

We work bottom-up for assignments...

\( \{ x = x_0 \land y = y_0 \} \)

\( t = x ; \)
\( \{ y = y_0 \land t = x_0 \} \quad P_3 \text{ is } \{ P_2[y/x] \} \)

\( x = y ; \)
\( \{ x = y_0 \land t = x_0 \} \quad \text{assignment} \)

\( y = t ; \)
\( \{ x = y_0 \land y = x_0 \} \quad \text{assignment} \)
Example: Composition of Assignments

To show: the following is satisfied under partial correctness.

We work bottom-up for assignments...

\[
\begin{align*}
&\begin{aligned}
&\begin{cases}
&x = x_0 \land y = y_0 \\
y = y_0 \land x = x_0
\end{cases} &\text{ (} P_3[x/t] \text{ )}
\end{aligned} \\
t = x ;
&\begin{aligned}
&\begin{cases}
&y = y_0 \land t = x_0
\end{cases} &\text{ assignment}
\end{aligned} \\
x = y ;
&\begin{aligned}
&\begin{cases}
&x = y_0 \land t = x_0
\end{cases} &\text{ assignment}
\end{aligned} \\
y = t ;
&\begin{aligned}
&\begin{cases}
&x = y_0 \land y = x_0
\end{cases} &\text{ assignment}
\end{aligned}
\end{align*}
\]
Example: Composition of Assignments

To show: the following is satisfied under partial correctness.

We work bottom-up for assignments...

\[ \begin{align*}
\{ x = x_0 \land y = y_0 \} \\
\{ y = y_0 \land x = x_0 \} & \quad \text{implied [proof required]} \\
t = x ; \\
\{ y = y_0 \land t = x_0 \} & \quad \text{assignment} \\
x = y ; \\
\{ x = y_0 \land t = x_0 \} & \quad \text{assignment} \\
y = t ; \\
\{ x = y_0 \land y = x_0 \} & \quad \text{assignment}
\end{align*} \]

Finally, show \( \{ x = x_0 \land y = y_0 \} \) implies \( \{ y = y_0 \land x = x_0 \} \).
Programs with Conditional Statements
Deduction Rules for Conditionals

if-then-else:

\[
\frac{(P \land B) \quad C_1 \quad QR \quad (P \land \neg B) \quad C_2 \quad QR}{(P) \quad \text{if } (B) \ C_1 \ \text{else } C_2 \ (QR)}
\]

(if-then-else)

if-then (without else):

\[
\frac{(P \land B) \quad C \quad QR \quad (P \land \neg B) \rightarrow QR}{(P) \quad \text{if } (B) \ C \ (QR)}
\]

(if-then)
Annotated program template for if-then-else:

\[
\begin{align*}
&P \quad \text{if-then-else} \\
&\{P \wedge B\} \quad \text{if ( } B \text{ ) } \{ \\
&\quad \{P \wedge \neg B\} \quad \text{if-then-else } \\
&C_1 \\
&\quad \{Q\} \quad (justify \ depending \ on \ C_1—which \ “subproof”\ ) \\
&\} \quad \text{else } \{ \\
&\quad \{P \wedge \neg B\} \quad \text{if-then-else } \\
&C_2 \\
&\quad \{Q\} \quad (justify \ depending \ on \ C_2—which \ “subproof”\ ) \\
&\} \\
\{Q\} \quad \text{if-then-else } [justifies \ this \ Q, \ given \ previous \ two]
\end{align*}
\]
Annotated program template for if-then:

\[
\begin{align*}
&P \\
\text{if } (B) \{ \\
&P \land B \\
&\text{if-then} \\
&C \\
&\text{[add justification based on } C]\}
\end{align*}
\]

\[
\begin{align*}
&Q \\
\text{if-then} \\
\text{Implied: Proof of } P \land \neg B \rightarrow Q
\end{align*}
\]
Example: Conditional Code

Example: Prove the following is satisfied under partial correctness.

\[
\begin{align*}
\langle \text{true} \rangle & \quad \langle P \rangle \\
\text{if ( } \max < x \text{ ) } \{ & \quad \text{if ( } B \text{ ) } \{ \\
& \quad \max = x ; \\
\} \quad C \\
\langle max \geq x \rangle & \quad \langle Q \rangle \\
\end{align*}
\]

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 “implies”:
   • 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.

```plaintext
\( \text{true} \)

if ( max < x ) {
    (\( \text{true} \land max < x \)) \hspace{1cm} \text{if-then}
    
    max = x ;
    (\( max \geq x \)) \hspace{1cm} \text{to be shown}
}

(\( max \geq x \)) \hspace{1cm} \text{if-then}

Implied: \( \text{true} \land \neg( max < x ) \) \( \rightarrow \) \( max \geq x \)
Doing the Steps

1. Add annotations for the if-then statement.
2. “Push up” for the assignments.

\[
\text{true}
\]

```java
if ( max < x ) {
    ( true ∧ max < x ) \hspace{1em} \text{if-then}
    ( x ≥ x )
    max = x ;
    ( max ≥ x ) \hspace{1em} \text{assignment}
}

( max ≥ x ) \hspace{1em} \text{if-then}
\]

Implied: \( ( true ∧ \neg ( max < x ) ) \rightarrow max ≥ x \)
Doing the Steps

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

```plaintext
\{ \text{true} \}

\text{if ( max < x )} \{ \\
  \{ \text{true} \land max < x \} \quad \text{if-then} \\
  \{ x \geq x \} \quad \text{Implied (a)} \\
  \text{max = x ;} \\
  \{ max \geq x \} \quad \text{assignment} \\
\}

\{ \text{max} \geq x \} \quad \text{if-then} \\
\hspace{1cm} \text{Implied (b): } (\text{true} \land \neg (max < x)) \rightarrow max \geq x
```
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):

\[ \vdash \left( (\text{true} \land (\text{max} < x)) \right) \rightarrow x \geq x. \]

Clearly \( x \geq x \) is a tautology and the implication holds.
Proof of Implied (b): Show \( \vdash (P \land \neg B) \rightarrow Q \), which is

\[ \vdash \left( \text{true} \land \neg (\text{max} < x) \right) \rightarrow (\text{max} \geq x) \].

1. \( \text{true} \land \neg (\text{max} < x) \) \hspace{1cm} \text{assumption}
2. \( \neg (\text{max} < x) \) \hspace{1cm} \land e: 1
3. \( \text{max} \geq x \) \hspace{1cm} \text{def. of } \geq
4. \( (\text{true} \land \neg (\text{max} < x)) \rightarrow (\text{max} \geq x) \) \hspace{1cm} \rightarrow i: 1–3
Example 2 for Conditionals

Prove the following is satisfied under partial correctness.

\[
\{ 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

```java
true

if ( x > y ) {
  x > y
  max = x ;
  (x > y ∧ max = x) ∨ (x ≤ y ∧ max = y)
}
else {
  ¬(x > y)
  max = y ;
  (x > y ∧ max = x) ∨ (x ≤ y ∧ max = y)
}

(x > y ∧ max = x) ∨ (x ≤ y ∧ max = y)

Program Verification  Conditionals
```
Example 2: Annotated Code

\[ \begin{array}{ll}
\text{true} & \\
\text{if} \ ( x > y ) \{ \\
\quad ( x > y ) & \text{if-then-else} \\
\quad ( x > y \land x = x ) \lor ( x \leq y \land x = y ) & \\
\text{max} = x ; & \text{assignment} \\
\quad ( x > y \land \text{max} = x ) \lor ( x \leq y \land \text{max} = y ) & \\
\} & \\
\text{else} \{ \\
\quad \neg ( x > y ) & \text{if-then-else} \\
\quad ( x > y \land y = x ) \lor ( x \leq y \land y = y ) & \\
\text{max} = y ; & \text{assignment} \\
\quad ( x > y \land \text{max} = x ) \lor ( x \leq y \land \text{max} = y ) & \\
\} & \\
\quad ( x > y \land \text{max} = x ) \lor ( x \leq y \land \text{max} = y ) & \text{if-then-else} \\
\end{array} \]
Example 2: Annotated Code

\( true \)

\[
\begin{align*}
\text{if } (x > y) \{ \\
\quad (x > y) \quad \text{if-then-else} \\
\quad (x > y \land x = x) \lor (x \leq y \land x = y) \quad \text{implied (a)} \\
\quad \text{max} = x ; \\
\quad (x > y \land \text{max} = x) \lor (x \leq y \land \text{max} = y) \quad \text{assignment} \\
\} \text{ else } \{ \\
\quad \neg(x > y) \quad \text{if-then-else} \\
\quad (x > y \land y = x) \lor (x \leq y \land y = y) \quad \text{implied (b)} \\
\quad \text{max} = y ; \\
\quad (x > y \land \text{max} = x) \lor (x \leq y \land \text{max} = y) \quad \text{assignment} \\
\}
\end{align*}
\]

\( (x > y \land \text{max} = x) \lor (x \leq y \land \text{max} = y) \)
Example 2: Implied Conditions

(a) Prove $x > y \rightarrow ((x > y \land x = x) \lor (x \leq y \land x = y))$.

1. $x > y$ \hspace{1cm} assumption
2. $x = x$ \hspace{1cm} EQ1
3. $x > y \land x = x$ \hspace{1cm} \land i: 1, 2
4. $(x > y \land x = x) \lor (x \leq y \land x = y)$ \hspace{1cm} \lor i: 3
5. $(x > y) \rightarrow (x > y \land x = x) \lor (x \leq y \land x = y)$ \hspace{1cm} \rightarrow i: 1–4

(b) Prove $x \leq y \rightarrow ((x > y \land y = x) \lor (x \leq y \land y = y))$.

1. $x \leq y$ \hspace{1cm} assumption
2. $y = y$ \hspace{1cm} EQ1
3. $x \leq y \land y = y$ \hspace{1cm} \land i: 1, 2
4. $(x > y \land y = x) \lor (x \leq y \land y = y)$ \hspace{1cm} \lor i: 3
5. $(x \leq y) \rightarrow ((x > y \land y = x) \lor (x \leq y \land y = y))$ \hspace{1cm} \rightarrow i: 1–4
While-Loops and Total Correctness
Inference Rule: Partial-while

“Partial while”: do not (yet) require termination.

\[
\frac{(I \land B) \quad C \quad (I)}{(I) \quad \text{while} \quad (B) \quad C \quad (I \land \neg B)} \quad \text{(partial-while)}
\]

In words:

If the code \(C\) satisfies the triple \((I \land B) \quad C \quad (I)\), 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

\[
\begin{align*}
\{ P \} \\
\{ I \} & \quad \text{Implied (a)} \\
\text{while} ( B ) \{ \\
\{ I \land B \} & \quad \text{partial-while} \\
C & \\
\{ I \} & \quad \leftarrow \text{to be justified, based on } C \\
\} \\
\{ I \land \neg B \} & \quad \text{partial-while} \\
\{ Q \} & \quad \text{Implied (b)}
\end{align*}
\]

(a) Prove \( P \rightarrow I \)  \hspace{1em} \text{(precondition } P \text{ implies the loop invariant)}

(b) Prove \( (I \land \neg B) \rightarrow Q \) \hspace{1em} \text{(exit condition implies postcondition)}

We need to determine \( I \)!!
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

\( \{ x \geq 0 \} \)

\( y = 1 ; \)

\( z = 0 ; \)

\( \text{while} \ (z \neq x) \{ \)
  \( z = z + 1 ; \)
  \( y = y \times z ; \)
\( \}

\( \{ y = x! \} \)
Example: Finding a loop invariant

\[ \{ x \geq 0 \} \]

\[ y = 1 ; \]
\[ z = 0 ; \]

\[ \rightarrow \text{while} \ (z \neq x) \{ \]
\[ \quad \quad z = z + 1 ; \]
\[ \quad \quad y = y \times z ; \]
\[ \} \]

\[ \{ y = x! \} \]

At the while statement:

\[
\begin{array}{cccc}
 x & y & z & z \neq x \\
 5 & 1 & 0 & \text{true} \\
 5 & 1 & 1 & \text{true} \\
 5 & 2 & 2 & \text{true} \\
 5 & 6 & 3 & \text{true} \\
 5 & 24 & 4 & \text{true} \\
 5 & 120 & 5 & \text{false} \\
\end{array}
\]
Example: Finding a loop invariant

\[
\begin{align*}
\{ x \geq 0 \} \\
y = 1 ; \\
z = 0 ; \\
\rightarrow \text{while (z }\neq x \} \\
& \quad z = z + 1 ; \\
& \quad y = y \times z ; \\
\} \\
\{ y = x! \}
\end{align*}
\]

At the while statement:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>z \neq x</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>true</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>true</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>2</td>
<td>true</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>3</td>
<td>true</td>
</tr>
<tr>
<td>5</td>
<td>24</td>
<td>4</td>
<td>true</td>
</tr>
<tr>
<td>5</td>
<td>120</td>
<td>5</td>
<td>false</td>
</tr>
</tbody>
</table>

From the trace and the postcondition, 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 \( (y = z!) \).

\[
\begin{align*}
\{ x \geq 0 \} & \\
y = 1 ; & \quad [\text{justification required}] \\
z = 0 ; & \quad \{ y = z! \} \\
\text{while } (z \neq x) \{ & \quad \text{partial-while } (\{ I \land B \}) \\
\{ (y = z!) \land \neg(z = x) \} & \\
z = z + 1 ; & \\
y = y \ast z ; & \quad [\text{justification required}] \\
\{ y = z! \} & \\
\} & \quad \text{partial-while } (\{ I \land \neg B \}) \\
\{ y = z! \land z = x \} & \\
\{ y = x! \}
\end{align*}
\]
Example: annotations for partial-while

Annotate assignment statements (bottom-up).

\[
\begin{align*}
(\begin{array}{c}
\text{x} \geq 0 \rangle \\
\text{1 = 0!} \rangle \\
y = 1 \\
\text{y = 0!} \rangle \\
z = 0 \\
\text{y = z!} \rangle \\
\text{while (z != x) {}} & \\
& \begin{array}{c}
(\begin{array}{c}
\text{(y = z!) } \land \neg (z = x) \rangle \\
\text{y(z + 1) = (z + 1)!} \rangle \\
z = z + 1 \\
\text{y = z!} \rangle \\
y = y \times z \\
\text{y = z!} \rangle
\end{array}
\end{array} \\
& \begin{array}{c}
\text{y = z!} \land z = x \rangle \\
\text{y = x!} \rangle
\end{array}
\end{align*}
\]

Program Verification while-Loops
Example: annotations for partial-while

Note the required implied conditions.

\(\{ x \geq 0 \}\)
\(\{ 1 = 0! \}\) \hspace{1cm} \text{implied (a)}
y = 1 ;
\(\{ y = 0! \}\) \hspace{1cm} \text{assignment}
z = 0 ;
\(\{ y = z! \}\) \hspace{1cm} \text{assignment}
while (z != x) {
\(\{ (y = z!) \land \neg(z = x) \}\) \hspace{1cm} \text{partial-while}
\(\{ y(z+1) = (z+1)! \}\) \hspace{1cm} \text{implied (b)}
z = z + 1 ;
\(\{ yz = z! \}\) \hspace{1cm} \text{assignment}
y = y * z ;
\(\{ y = z! \}\) \hspace{1cm} \text{assignment}
}
\(\{ y = z! \land z = x \}\) \hspace{1cm} \text{partial-while}
\(\{ y = x! \}\) \hspace{1cm} \text{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): \((y = z!) \land (z = x) \vdash (y = x!)\).

1. \((y = z!) \land (z = x)\)  premise
2. \(y = z!\)  \&e: 1
3. \(z = x\)  \&e: 1
4. \(y = x!\)  EqSubs: 2, 3
Example: implied condition (b)

Proof of implied (b):

\[ ((y = z!) \land \neg(z = x)) \vdash (z + 1) \, y = (z + 1)! \]

1. \( y = z! \land z \neq x \) \hspace{1cm} \text{premise}
2. \( y = z! \) \hspace{1cm} \land\!e: 1
3. \( (z + 1) \, y = (z + 1) \, z! \) \hspace{1cm} \text{EqSubs: 2}
4. \( (z + 1) \, z! = (z + 1)! \) \hspace{1cm} \text{def. of factorial}
5. \( (z + 1) \, y = (z + 1)! \) \hspace{1cm} \text{EqTrans: 3, 4}
Example 2 (Partial-while)

Prove the following is satisfied under partial correctness.

\[
\begin{align*}
\{ n \geq 0 \land a \geq 0 \} \\
\text{s} = 1 \\
\text{i} = 0 \\
\rightarrow \text{while } (\text{i} < \text{n}) \{ \\
\quad \text{s} = \text{s} \times a \\
\quad \text{i} = \text{i} + 1 \\
\} \\
\{ \text{s} = a^n \}
\end{align*}
\]
Example 2 (Partial-while)

Prove the following is satisfied under partial correctness.

\[ \{ \ n \geq 0 \land a \geq 0 \ \} \]

\[
\begin{align*}
  s &= 1 ; \\
  i &= 0 ; \\
  \longrightarrow & \text{while (i < n) } \{ \\
  & \quad s = s \times a ; \\
  & \quad i = i + 1 ; \\
  \} \\
\[ \{ s = a^n \} \]
\end{align*}
\]

Trace of the loop:

<table>
<thead>
<tr>
<th>a</th>
<th>n</th>
<th>i</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1*2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1<em>2</em>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>1<em>2</em>2*2</td>
</tr>
</tbody>
</table>

Candidate for the loop invariant:

\[ s = a^i \]
Example 2 (Partial-while)

Prove the following is satisfied under partial correctness.

\[ \langle n \geq 0 \land a \geq 0 \rangle \]

\[ s = 1 ; \]

\[ i = 0 ; \]

\[ \rightarrow \text{while } (i < n) \{ \]

\[ s = s * a ; \]

\[ i = i + 1 ; \]

\[ \} \]

\[ \langle s = a^n \rangle \]

Trace of the loop:

<table>
<thead>
<tr>
<th>a</th>
<th>n</th>
<th>i</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1*2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1<em>2</em>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>1<em>2</em>2*2</td>
</tr>
</tbody>
</table>

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.

\[
\begin{align*}
&\text{( } n \geq 0 \land a \geq 0 \text{ ) } \\
&\text{( } ... \text{ ) } \\
&s = 1 ; \\
&\text{( } ... \text{ ) } \\
&i = 0 ; \\
&\text{( } s = a^i \text{ ) } \\
&\text{while ( } i < n \text{ ) } \{ \\
&\text{( } s = a^i \land i < n \text{ ) partial-while } \\
&\text{( } ... \text{ ) } \\
&s = s \times a ; \\
&\text{( } ... \text{ ) } \\
&i = i + 1 ; \\
&\text{( } s = a^i \text{ ) } \\
&\} \\
&\text{( } s = a^i \land i \geq n \text{ ) partial-while } \\
&\text{( } s = a^n \text{ ) implied (c) } \\
\end{align*}
\]
**Example 2: Adjusted invariant**

Try a new invariant:

\[ s = a^i \land i \leq n \] .

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

| \( \{ n \geq 0 \land a \geq 0 \} \) | implied (a) |
| \( \{ 1 = a^0 \land 0 \leq n \} \) | |
|  \( s = 1 ; \) | assignment |
| \( \{ s = a^0 \land 0 \leq n \} \) | |
| \( i = 0 ; \) | assignment |
| \( \{ s = a^i \land i \leq n \} \) | |

while (\( i < n \)) {

\( \{ s = a^i \land i \leq n \land i < n \} \) | partial-while |
\( \{ s \cdot a = a^{i+1} \land i + 1 \leq n \} \) | implied (b) |
\( s = s \ast a ; \) | assignment |
\( \{ s = a^{i+1} \land i + 1 \leq n \} \) | |
\( i = i + 1 ; \) | assignment |
\( \{ s = a^i \land i \leq n \} \) | |

} | partial-while |

\( \{ s = a^i \land i \leq n \land i \geq n \} \) | implied (c) |
\( \{ s = a^n \} \) |
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.
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 \geq 0)
y = 1;
z = 0;

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

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

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

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

$x - z$ decreases by 1

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

\[
\begin{align*}
&\{ x \geq 0 \} \\
y = 1 ; \\
z = 0 ;
\end{align*}
\]

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

```python
while ( z != x ) {
    z = z + 1 ;  \quad x - z \text{ decreases by 1}
    y = y * z ; \quad x - z \text{ unchanged}
}\}
\[
\{ y = x! \}
```

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

\[
\begin{align*}
&\langle x \geq 0 \rangle \\
y = 1 ; \\
z = 0 ; \\
\text{At start of loop: } x - z = x \geq 0
\end{align*}
\]

\[
\text{while ( z ! = x ) }
\begin{aligned}
z &= z + 1 ; & x - z \text{ decreases by 1} \\
y &= y * z ; & x - z \text{ unchanged}
\end{aligned}
\]

\[
\langle y = x! \rangle
\]

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 \textit{variant}).

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

- \textbf{Precondition:} $x \geq 0$.
- \textbf{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 $z \neq x$ ends the loop.
Arrays
Assignment of Values of an Array


Assignment may work as before:

$$\langle P[A[x]/v] \rangle$$

$v = A[x]$ ;

$$\langle P \rangle$$ assignment

But a complication can occur:

$$\langle A[y] = 0 \rangle$$

$A[x] = 1$;

$$\langle A[y] = 0 \rangle$$

The conclusion is not valid if $x = y$.

A correct rule must account for possible changes to $A[y], A[z+3]$, etc., when $A[x]$ changes.
Our solution: Treat an assignment to an array value

\[ A[e_1] = e_2 \]

as an assignment of the whole array:

\[ A = A\{e_1 \leftarrow e_2\} ; \]

where the term “\( A\{e_1 \leftarrow e_2\} \)” denotes an array identical to \( A \) except that the \( e_1^{th} \) element is changed to have the value \( e_2 \).
Array Assignment: Definition and Examples

**Definition:** $A\{i \leftarrow e\}$ denotes the array with entries given by

$$A\{i \leftarrow e\}[j] = \begin{cases} e, & \text{if } j = i \\ A[j], & \text{if } j \neq i \end{cases}$$

**Examples:**

$$A\{1 \leftarrow 7\}{\{2 \leftarrow 14\}}[2] = ??$$

$$A\{1 \leftarrow 7\}{\{2 \leftarrow 14\}}{\{3 \leftarrow 21\))[2] = ??$$

$$A\{1 \leftarrow 7\}{\{2 \leftarrow 14\}}{\{3 \leftarrow 21\}}[i] = ??$$
The Array-Assignment Rule

Array assignment:

\[
\left( Q[A\{e_1 \leftarrow e_2\}/A] \right) A[e_1] = e_2 \quad (Q)
\]

where

\[
A\{i \leftarrow e\}[j] = \begin{cases} 
  e, & \text{if } j = i \\
  A[j], & \text{if } j \neq i
\end{cases}
\]
Example

Prove the following is satisfied under partial correctness.

$$\{ A[x] = x_0 \land A[y] = y_0 \}$$

\[ t = A[x] ; \]
\[ A[x] = A[y] ; \]
\[ A[y] = t ; \]
\[ \{ A[x] = y_0 \land A[y] = x_0 \} \]

We do assignments bottom-up, as always....
Example: push up assertions for assignments

\[
\begin{align*}
(A[x] = x_0 \land A[y] = y_0) \\\nt = A[x] ; \\
(A[y ← t][x] = y_0 \land A[y ← t][y] = x_0) \\
A[y] = t ; \\
(A[x] = y_0 \land A[y] = x_0) \quad \text{array assignment}
\end{align*}
\]
Example: push up assertions for assignments

\[
\begin{aligned}
&\langle A[x] = x_0 \land A[y] = y_0 \rangle \\
&\text{t} = A[x] ; \\
&\langle A\{x \leftarrow A[y]\}\{y \leftarrow t\}[x] = y_0 \\
&\quad \land A\{x \leftarrow A[y]\}\{y \leftarrow t\}[y] = x_0 \rangle \\
&\langle A\{y \leftarrow t\}[x] = y_0 \land A\{y \leftarrow t\}[y] = x_0 \rangle
\end{aligned}
\]

array assignment

\[
\begin{aligned}
&A[y] = t ; \\
&\langle A[x] = y_0 \land A[y] = x_0 \rangle
\end{aligned}
\]

array assignment
Example: push up assertions for assignments

\[ \begin{align*}
& \langle A[x] = x_0 \land A[y] = y_0 \rangle \\
& \langle A\{x \leftarrow A[y]\}\{y \leftarrow A[x]\}\}[x] = y_0 \\
& \quad \land A\{x \leftarrow A[y]\}\{y \leftarrow A[x]\}\}[y] = x_0 \rangle \\
& \text{t} = A[x] ; \\
& \langle A\{x \leftarrow A[y]\}\{y \leftarrow t\}\}[x] = y_0 \\
& \quad \land A\{x \leftarrow A[y]\}\{y \leftarrow t\}\}[y] = x_0 \rangle \\
& \langle A\{y \leftarrow t\}\}[x] = y_0 \land A\{y \leftarrow t\}\}[y] = x_0 \rangle \\
& A[y] = t ; \\
& \langle A[x] = y_0 \land A[y] = x_0 \rangle
\end{align*} \]
Example: push up assertions for assignments

\[ A[x] = x_0 \land A[y] = y_0 \]
\[ A\{x \leftarrow A[y]\}\{y \leftarrow A[x]\}[x] = y_0 \]
\[ \land A\{x \leftarrow A[y]\}\{y \leftarrow A[x]\}[y] = x_0 \]  
\[ \text{implied (a)} \]

t = A[x] ;
\[ A\{x \leftarrow A[y]\}\{y \leftarrow t\}[x] = y_0 \]
\[ \land A\{x \leftarrow A[y]\}\{y \leftarrow t\}[y] = x_0 \]  
\[ \text{assignment} \]

\[ A\{y \leftarrow t\}[x] = y_0 \land A\{y \leftarrow t\}[y] = x_0 \]  
\[ \text{array assignment} \]

A[y] = t ;
\[ A[x] = y_0 \land A[y] = x_0 \]  
\[ \text{array assignment} \]
Example: Proof of implied

As “implied (a)”, we need to prove the following.

**Lemma:**

\[ A\{x \leftarrow A[y]\}\{y \leftarrow A[x]\}[x] = A[y] \]

and

\[ A\{x \leftarrow A[y]\}\{y \leftarrow A[x]\}[y] = A[x] \]

**Proof.**

In the second equation, the index element is the assigned element.

For the first equation, we consider two cases.

- If \( y \neq x \), the “\( \{y \leftarrow \ldots\} \)” is irrelevant, and the claim holds.
- If \( y = x \), the result on the left is \( A[x] \), which is also \( A[y] \).
Example: Alternative proof

For an alternative proof, use the definition of $M\{i \leftarrow e\}[j]$, with $A\{x \leftarrow A[y]\}$ as $M$, $i = y$ and $e = A[x]$:

$$A\{x \leftarrow A[y]\}\{y \leftarrow A[x]\}[j] = \begin{cases} A[x], & \text{if } y = j \\ A\{x \leftarrow A[y]\}[j], & \text{if } y \neq j \end{cases}.$$

At index $j = y$, this is just $A[x]$, as required.

In the case $j = x$, we get the required value $A[y]$. (Why?)

And, finally, if $j \neq x$ and $j \neq y$, then

$$A\{x \leftarrow A[y]\}\{y \leftarrow A[x]\}[j] = A[j],$$

as we should have required.
Example: reversing an array

Example: Given an array $R$ with $n$ elements, reverse the elements.

Algorithm: exchange $R[j]$ with $R[n + 1 - j]$, for each $1 \leq j \leq \lfloor n/2 \rfloor$.

A possible program is

```c
j = 1;
while ( 2*j <= n ) {
    t = R[j] ;
    R[j] = R[n+1-j] ;
    R[n+1-j] = t ;
    j = j + 1 ;
}
```

Needed: a postcondition, and a loop invariant.
Reversal code: conditions and an invariant

Precondition: \( (\forall x ((1 \leq x \leq n) \rightarrow (R[x] = r_{x})) \) \).

Postcondition: \( (\forall x ((1 \leq x \leq n) \rightarrow (R[x] = r_{n+1-x})) \) \).

Invariant? When exchanging at position \( j \)?

- If \( x < j \) or \( x > n + 1 - j \), then \( R[x] \) and \( R[n + 1 - x] \) have already been exchanged.
- If \( j \leq x \leq n + 1 - j \), then no exchange has happened yet.

Thus let \( Inv'(j) \) be the formula

\[
(\forall x \left( ((1 \leq x < j) \rightarrow (R[x] = r_{n+1-x} \land R[n + 1 - x] = r_{x})) \right.
\]

\[
\left. \land ((j \leq x \leq n/2) \rightarrow (R[x] = r_{x} \land R[n + 1 - x] = r_{n+1-x})) \right) .
\]

and \( Inv(j) = Inv'(j) \land (1 \leq j \leq n) \).
Reversal: Annotations around the loop

The annotations surrounding the while-loop:

\[
\left( (n \geq 0) \land \left( \forall x \left( (1 \leq x \leq n) \rightarrow (R[x] = r_x) \right) \right) \right) \}
\left( INV(1) \right) \}

\begin{array}{ll}
\{ j = 1 ; \\
\{ INV(j) \}
\end{array}

while ( 2*j <= n ) {
\begin{array}{ll}
\{ (INV(j) \land (2j \leq n)) \}
\{ INV(j) \}
\end{array}

\begin{array}{ll}
\} \\
\{ (INV(j) \land (2j > n)) \}
\{ INV(j) \}
\end{array}

\left( \forall x \left( (1 \leq x \leq n) \rightarrow (R[x] = r_{n+1-x}) \right) \right) \}

\text{implied (a)}

\text{assignment}

\text{partial-while}

(TBA)

\text{partial-while}

\text{implied (b)}
Reversal code: annotations inside the loop

We must now handle the code inside the loop.

\[
\begin{align*}
\{( \text{Inv}(j) \land 2j \leq n) \} \\
\{( \text{Inv}(j + 1)[R'/R], \text{where } R' \text{ is} \\
\quad R\{j \leftarrow R[n + 1 - j]\}\{ (n + 1 - j) \leftarrow R[j]\}\} \\
t = R[j]; R[j] = R[n+1-j]; R[n+1-j] = t; \\
\{( \text{Inv}(j + 1) \} \\
\}
\end{align*}
\]

Lemma

\[
\begin{align*}
\}
\end{align*}
\]

Assignment

\[
\begin{align*}
\}
\end{align*}
\]

Lemma

\[
\begin{align*}
\}
\end{align*}
\]
Proof of Implied Condition (c)

Recall $\text{Inv'}(j)$:

$$\left( \forall x \left( \left( (1 \leq x < j) \rightarrow (R[x] = r_{n+1-x} \land R[n+1-x] = r_{n+1}) \right) \right)$$

$$\quad \land \left( (j \leq x \leq n/2) \rightarrow (R[x] = r_{n+1-x} \land R[n+1-x] = r_{n+1-x}) \right) \right) .$$

We need this to imply $\text{Inv'}(j+1)[R'/R]$, which is

$$\left( \forall x \left( \left( (1 \leq x < j + 1) \rightarrow (R'[x] = r_{n+1-x} \land R'[n+1-x] = r_{n+1}) \right) \right)$$

$$\quad \land \left( (j + 1 \leq x \leq n/2) \rightarrow (R'[x] = r_{n+1-x} \land R'[n+1-x] = r_{n+1-x}) \right) \right) ,$$

which by the construction of $R'$ is equivalent to

$$\left( \forall x \left( \left( (1 \leq x < j) \rightarrow (R[x] = r_{n+1-x} \land R[n+1-x] = r_{n+1}) \right) \right)$$

$$\quad \land (R'[j] = r_{n+1-j}) \land (R'[n+1-j] = r_{n+1-j})$$

$$\quad \land \left( (j + 1 \leq x \leq n/2) \rightarrow (R[x] = r_{n+1-x} \land R[n+1-x] = r_{n+1-x}) \right) \right) .$$
Example: Binary Search
Binary Search

Binary search is a very common technique, to find whether a given item exists in a sorted array.

Although the algorithm is simple in principle, it is easy to get the details wrong. Hence verification is in order.

Inputs: Array $A$ indexed from 1 to $n$; integer $x$.

Precondition: $A$ is sorted: $\forall i \forall j \left( (1 \leq i < j \leq n) \rightarrow (A[i] \leq A[j]) \right)$.

Output values: boolean $\text{found}$; integer $m$.

Postcondition: Either $\text{found}$ is true and $A[m] = x$, or $\text{found}$ is false and $x$ does not occur at any location of $A$.

(Also, $A$ and $x$ are unchanged; We simply won’t write to either.)
Code: The outer loop

\[
\forall i \forall j \left( (1 \leq i < j \leq n) \rightarrow (A[i] \leq A[j]) \right)
\]

\( l = 1; \quad u = n; \quad \text{found} = \text{false}; \)

\( \langle I \rangle \)

while (\ l \leq u \ \text{and} \ \neg \ \text{found} \ ) \{ \\
\quad \langle I \wedge (l \leq u \wedge \neg \ \text{found}) \rangle \)

\( m = (l+u) \div 2; \)

\( \langle J \rangle \)

if (A[m] = x) {

\quad \text{...Body omitted...}

}

\( \langle I \rangle \)

\( \langle I \wedge \neg(l \leq u \wedge \neg \ \text{found}) \rangle \)

\( \langle (\ \text{found} \wedge A[m] = x) \ \lor \ (\neg \ \text{found} \wedge \forall k \ \neg(A[k] = x)) \rangle \)
Program Verification
Example: Binary Search

\( J \)

\( \text{if} \ ( A[m] = x ) \) 

\( J \land (A[m] = x) \) \quad \text{if-then-else}

\( \text{found} = \text{true}; \)

\( I \)

\( \text{else if} \ ( A[m] < x ) \) 

\( J \land \neg(A[m] = x) \land (A[m] < x) \) \quad \text{if-then-else}

\( l = m + 1; \)

\( I \)

\( \text{else} \) 

\( J \land \neg(A[m] = x) \land \neg(A[m] < x) \) \quad \text{if-then-else}

\( u = m - 1; \)

\( I \)

\( I \) \quad \text{if-then-else}
Invariant for Binary Search

In the loop, there are two cases:

- We have found the target, at position $m$.
- We have not yet found the target; if it is present, it must lie between $A[\ell]$ and $A[u]$ (inclusive).

Expressed as a formula:

$$
(\text{found} \land A[m] = x) \lor (\neg\text{found} \land \forall i \ ((A[i] = x) \rightarrow (\ell \leq i \leq u)))
$$

It turns out that the above is more specific than necessary. As the actual invariant, we shall use the formula

$$
I = (\text{found} \rightarrow A[m] = x) \land (\forall i \ ((A[i] = x) \rightarrow (\ell \leq i \leq u)))
$$

(Exercise: As you go through the proof, check what would happen if we used the first formula instead.)
Annotations for while

\( \forall i \forall j ((1 \leq i < j \leq n) \rightarrow (A[i] \leq A[j])) \)

\( l = 1; \ u = n; \ \text{found} = \text{false}; \)

\( (\text{found} \rightarrow A[m] = x) \land (\forall i ((A[i] = x) \rightarrow (\ell \leq i \leq u))) \)

while ( \( l \leq u \land \neg \text{found} \) ) {

\( (\text{found} \rightarrow A[m] = x) \land (\forall i ((A[i] = x) \rightarrow (\ell \leq i \leq u))) \land (\ell \leq u) \land \neg \text{found} \)

m = (l+u) div 2 ;

\( (\forall i ((A[i] = x) \rightarrow (\ell \leq i \leq u)) \land \neg \text{found} \land (\ell \leq m \leq u) \)

The last condition is the formula “\(J\)”: the precondition for the if-statement.

Program Verification  Example: Binary Search  396/434
First Branch of the if-Statement

\[
\begin{align*}
\{ \forall i \left( (A[i] = x) \rightarrow (\ell \leq i \leq u) \right) \land \neg \text{found} \land (\ell \leq m \leq u) \} \\
\text{if} \ (A[m] = x) \ {\{} \\
\{ \forall i \left( (A[i] = x) \rightarrow (\ell \leq i \leq u) \right) \land \neg \text{found} \land (\ell \leq m \leq u) \land (A[m] = x) \} \\
\text{if-then-else} \\
\{ \left( \text{true} \rightarrow A[m] = x \right) \land \left( \forall i \left( (A[i] = x) \rightarrow (\ell \leq i \leq u) \right) \right) \} \\
\text{implied} \\
\text{found} = \text{true}; \\
\{ \left( \text{found} \rightarrow A[m] = x \right) \land \left( \forall i \left( (A[i] = x) \rightarrow (\ell \leq i \leq u) \right) \right) \} \\
\text{assignment} \\
\}
\end{align*}
\]

The implication is trivial.
Second Branch of the if-Statement

\[ \forall i \left( (A[i] = x) \rightarrow (\ell \leq i \leq u) \right) \land \neg found \land (\ell \leq m \leq u) \land \neg (A[m] = x) \] 

if ( A[m] < x ) {

    \( \forall i \left( (A[i] = x) \rightarrow (\ell \leq i \leq u) \right) \land \neg found \land (\ell \leq m \leq u) \land \neg (A[m] = x) \land (A[m] < x) \) if-then-else

    \( (found \rightarrow A[m] = x) \land \left( \forall i \left( (A[i] = x) \rightarrow (m + 1 \leq i \leq u) \right) \right) \) implied

    l = m + 1;

    \( (found \rightarrow A[m] = x) \land \left( \forall i \left( (A[i] = x) \rightarrow (\ell \leq i \leq u) \right) \right) \) assignment

} 

To justify the implication, show that \( A[j] < x \) whenever \( \ell \leq j \leq m \).

This follows from the condition that \( A \) is sorted, together with \( A[m] < x \).
An Extended Example: Sorting
Suppose the code $C_{\text{sort}}$ is intended to sort $n$ elements of array $A$.

Give pre- and postconditions for $C_{\text{sort}}$, using a predicate $\text{sorted}(A, n)$ which is true iff $A[1] \leq A[2] \leq \ldots \leq A[n]$.

**First Attempt**

$\{ n \geq 1 \}$

$C_{\text{sort}}$

$\{ \text{sorted}(A, n) \}$
**Postcondition for Sorting**

Suppose the code $C_{\text{sort}}$ is intended to sort $n$ elements of array $A$.

Give pre- and postconditions for $C_{\text{sort}}$, using a predicate $\text{sorted}(A, n)$ which is true iff $A[1] \leq A[2] \leq \ldots \leq A[n]$.

**First Attempt**

\[
\begin{align*}
\{ n \geq 1 \} & \quad \{ n \geq 1 \} \\
& \quad \text{for } i = 1 \text{ to } n \{ \\
C_{\text{sort}} & \quad A[i] = 0 ; \\
& \quad \} \\
\{ \text{sorted}(A, n) \} & \quad \{ \text{sorted}(A, n) \}
\end{align*}
\]
Postcondition for Sorting, II

Let $\text{permutation}(A, A', n)$ mean that array $A[1], A[2], \ldots, A[n]$ is a permutation of array $A'[1], A'[2], \ldots, A'[n]$.

($A'$ will be a logical variable, not a program variable.)

Second Attempt

$$\{ n \geq 1 \land A = A' \}$$

$C_{\text{sort}}$

$$\{ \text{sorted}(A, n) \land \text{permutation}(A, A', n) \}$$
Let $\text{permutation}(A, A', n)$ mean that array $A[1], A[2], \ldots, A[n]$ is a permutation of array $A'[1], A'[2], \ldots, A'[n]$.

($A'$ will be a logical variable, not a program variable.)

**Second Attempt**

\[
\begin{align*}
&\left( n \geq 1 \land A = A' \right) \\
&\left( n \geq 1 \land A = A' \right) \\
&C_{\text{sort}} \quad \text{some algorithm on } A; \\
&n = 1; \\
&\left( \text{sorted}(A, n) \land \right. \\
&\left. \text{permutation}(A, A', n) \right) \\
&\left( \text{sorted}(A, n) \land \text{permutation}(A, A', n) \right)
\end{align*}
\]
Final Attempt (Correct)

\[
\begin{align*}
\langle n \geq 1 \land n = n_0 \land A = A' \rangle \\
C_{\text{sort}} \\
\langle \text{sorted}(A, n_0) \land \text{permutation}(A, A', n_0) \rangle
\end{align*}
\]
We shall briefly describe two algorithms for sorting.

- Insertion Sort
- Quicksort

Each has an "inner loop" which we will then consider.
Overview of Insertion Sort


Plan: insert each element, in turn, into the array of previously sorted elements.

Algorithm:

At the start, $A[1]$ is sorted (as an array of length 1). For each $k$ from 2 to $n$

Assume the array is sorted up to position $k - 1$


Compare it with $A[k - 1], A[k - 2], \ldots$, until its proper place is reached.
Insertion Sort: Inserting one element

Possible code for the insertion loop:

```java
i = k ;
while ( i > 1 ) {
    if ( A[i] < A[i-1] ) {
        t = A[i] ;
        A[i-1] = t ;
    }
    i = i - 1 ;
}
```

For correctness of this code, see the current assignment.
Overview of Quicksort

Quicksort is an ingenious algorithm, with many variations. Sometimes it works very well, sometimes not so well. We shall ignore most of those issues, however, and just look at a central step of the algorithm.

Idea:

Select one element of the array, called the *pivot*. (Which one? A complicated issue. YMMV.)

Separate the array into two parts: those less than or equal to the pivot, and those greater than the pivot.

Recursively sort each of the two parts.

Here, we shall focus on the middle step: “partition” the array according to the chosen pivot.
Partitioning an Array

Given: Array $X$ of length $n$, and a pivot $p$.

Goal: Put the “small” elements (those less than or equal to $p$) to the left part of the array, and the “large” elements (those greater than $p$) to the right.

Plan: Scan the array. Upon finding a large element appearing before a small element, exchange the two.

Requisite: Do all exchanges in a single scan. (Linear time!)
Partition: The Algorithm

Idea: keep the array elements in three sections of the array.

- Those known to be small (less than or equal to the pivot).
- Those known to be large (larger than the pivot).
- Unknown elements (not yet examined).

We mark the separations with pointers (indices) $a$ and $b$, as shown.

\[
\begin{array}{c|c|c}
\text{small} & \text{large} & \text{unknown} \\
\hline
a & b
\end{array}
\]

One step of the algorithm:

- If $X[b]$ is small, swap it with $X[a]$ and increment $a$.
- Increment $b$. 
Code and Pre- and Postconditions

\[
\begin{align*}
& (\ n \geq 1 \ ) \\
& a = 1 \\
& \text{while ( } a < n \ \&\& \ X[a] \leq p \ ) \{ \\
& \quad a = a + 1 \\
& \} \\
& b = a + 1 \\
& \text{while ( } b \leq n \ ) \{ \\
& \quad \text{if ( } X[b] \leq p \ ) \{ \\
& \quad \quad t = X[b] \ ; \ X[b] = X[a] \ ; \ X[a] = t \\
& \quad \quad a = a + 1 \\
& \quad \} \\
& \quad b = b + 1 \\
& \}
\end{align*}
\]

\[
(\exists z \ (1 \leq z \leq n + 1) \land (X[1..z] \leq p) \land (X[z..n] > p))
\]

Notation: “\(X[j..k]\)…” means “\(X[i]\)…, for each \(j \leq i < k\)”. “\(X[j..k]\)…” means “\(X[i]\)…, for each \(j \leq i \leq k\)”.
Annotation: First loop

Desired postcondition for the first loop:

\[
¬\left( (X[1..a) \leq p) \land ((a \geq n) \lor (X[a] > p)) \right) \]

Annotation for the while, and pushing up, yields:

\[
\left( (X[1..1) \leq p) \right)
\]

\[a = 1 ;\]

\[
\left( (X[1..a) \leq p) \right)
\]

while ( a < n && X[a] <= p ) {

\[
\left( (X[1..a) \leq p) \land ((a < n) \land (X[a] \leq p)) \right)
\]

\[
\left( (X[1..a + 1) \leq p) \right)
\]

\[a = a + 1 ;\]

\[
\left( (X[1..a) \leq p) \right)
\]

}\n
\[
\left( (X[1..a) \leq p) \land ((a \geq n) \lor (X[a] > p)) \right)
\]
The Second while-Loop

For the second while-loop, a good candidate for the invariant is

\[(X[1..a) \leq p) \land (X[a..b) > p)\] .

Let’s see if this works....

Colour key:

- **Greenish**: lower part of the array
- **Blueish**: upper part of the array
- **Either, reddened**: result of a substitution
- **Reddish**: condition from guards
“And” the loop guard to the invariant at the start of the loop. Then pushing up through the assignment and if yields

\[
\begin{aligned}
&\{ (X[1..a] \leq p) \land (X[a..b] > p) \land (b \leq n) \} \\
&\text{if } (X[b] \leq p) \{ \\
&\quad \{ (X[1..a] \leq p) \land (X[a..b] > p) \land (b \leq n) \land X[b] \leq p \} \\
&\quad \text{if-then} \\
&\}}
\end{aligned}
\]

\[
\begin{aligned}
&\} \\
&\quad \{ (X[1..a] \leq p) \land (X[a..b + 1] > p) \} \\
&\quad \text{if-then + implied} \\
\}
\end{aligned}
\]

\[
\begin{aligned}
&b = b + 1 \\
&\{ (X[1..a] \leq p) \land (X[a..b] > p) \} \\
&\text{assignment}
\end{aligned}
\]
Inside the if-Statement

Push up for the assignments inside the if:

\[
\text{if-then } \left( (X[1..a) \leq p) \land (X[a..b) > p) \land (b \leq n) \land X[b] \leq p \right) \\
\text{implied } \left( ((X[1..a) \leq p)) \land (X[a] > p) \land (X[a+1..b) > p) \land X[b] \leq p \right) \\
\text{swap } \left( (X[1..a+1) \leq p) \land (X[a+1..b+1) > p) \right) \\
\text{implied } a = a + 1 ; \\
\text{assignment } \left( (X[1..a) \leq p) \land (X[a..b+1) > p) \right)
\]

(The extra “implied” just makes the “swap” clearer.)
Putting It All Together

The annotation thus far works fine. But there is a “glitch”...

Between the loops, we have

\[
\begin{align*}
\mathcal{I} & \ (X[1..a) \leq p) \land ((a \geq n) \lor (X[a] > p)) \ \triangleright \partial \text{ partial-while} \\
\mathcal{I} & \ (X[1..a) \leq p) \land (X[a..a + 1) > p) \ \triangleright \partial \text{ implied} \\
b & = a + 1 \ ; \\
\mathcal{I} & \ (X[1..a) \leq p) \land (X[a..b) > p) \ \triangleright \partial \text{ assignment}
\end{align*}
\]

But the “implied” fails in the case that the first loop ended with \(a = n\) — we can’t deduce \(X[a] > p\).

Solution: either

- add an extra test to the code, or
- add \(1 \leq a \leq n\) to the first invariant, and modify the second to \((X[1..a) \leq p) \land ((a = n) \lor (X[a..b) > p))\).
Second while-Loop: Full annotation

\[ \langle (1 \leq a \leq n) \land (X[1..a) \leq p) \land ((a = n) \lor (X[a] > p)) \rangle \] partial-while
\[ \langle X[1..a) \leq p \rangle \land ((a = n) \lor (X[a..a + 1) > p)) \] implied
\[ b = a + 1 ; \]
\[ \langle X[1..a) \leq p \rangle \land ((a = n) \lor (X[a..b) > p) \rangle \] assignment
while ( b \leq n ) {
\[ \langle X[1..a) \leq p \rangle \land ((a = n) \lor (X[a..b) > p) \rangle \land (b \leq n) \] partial-while
\[ \langle X[1..a) \leq p \rangle \land (X[a..b) > p) \land (b \leq n) \] implied
if (X[b] \leq p ) {
}
\[ \langle X[1..a) \leq p \rangle \land (X[a..b) > p) \] if-then
\[ \langle X[1..a) \leq p \rangle \land ((a = n) \lor (X[a..b) > p) \rangle \] implied
}
\[ \langle X[1..a) \leq p \rangle \land ((a = n) \lor (X[a..b) > p) \rangle \land (b > n) \] partial-while
\[ \exists z (1 \leq z \leq n + 1) \land (X[1..z) \leq p) \land (X[z..n] > p) \] implied

"Implied" proofs left to you.