Program Verification

Outline

- Introduction: What and Why?
- Pre- and Post-conditions
- Conditionals
- `while`-Loops and Total Correctness
- `for`-Loops
- Arrays
- Termination (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 (white box, black box)
  - *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, never their absence."

[E. Dijkstra, 1972.]

Program 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 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

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

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 considering **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.
- Statements 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;
}
```

Example

```
    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

Example

```
    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.

Specifications

Example.

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

What if $x = -4$?

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.

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.

Specification of a Program

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

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

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 ;
```

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 FOL, 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

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

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

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

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. Total correctness satisfied:

$(\!| x = 1 |\!)$
y=x;
$(\!| y = 1 |\!)$

Example 2. Neither total nor partial correctness:

$(\!| x = 1 |\!)$
y=x ;
$(\!| y = 2 |\!)$

Examples for Partial and Total Correctness

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

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

What happens if we remove the pre-condition?

Examples for Partial and Total Correctness

Example 5. No correctness, because input altered ("consumed")

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

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

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

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;
}
```

$(\!| \, y = x_0! \, |\!)$

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;
}
```

$(\!| \, y = x_0! \, |\!)$

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.

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)

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)

> Rules for each construct in our programming language.

Presentation of a Proof

A full proof will have one or more Hoare triples before and after each code statement. Each triple 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 was true of that value beforehand.

Assignment: Example

Example.

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

by one application of the assignment rule.

More Examples for Assignment

Example 1.

$(\!\| y = 2 \|\!)$	$(\!\| P[E/x] \|\!)$
x = y ;	$x = E;$
$(\!\| x = 2 \|\!)$	$(\!\| P \|\!)$

Example 2.

$(\!\| 0 < 2 \|\!)$	$(\!\| P[E/x] \|\!)$
x = 2 ;	$x = E;$
$(\!\| 0 < x \|\!)$	$(\!\| P \|\!)$

Examples of Assignment

Example 3.

$\{\!\mid x+1=2 \mid\!\}$	$\{\!\mid (x=2)[(x+1)/x] \mid\!\}$
`x = x + 1;`	$x = x+1;$
$\{\!\mid x=2 \mid\!\}$	$\{\!\mid x=2 \mid\!\}$

Example 4.

$\{\!\mid x+1=n+1 \mid\!\}$
`x = x + 1;`
$\{\!\mid x=n+1 \mid\!\}$

Note about Examples

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

??	$(\!\| P[E/x] \|\!)$
x = y;	$x = E;$
$(\!\| x > 0 \|\!)$	$(\!\| P \|\!)$

Inference Rules with Implications

Precondition strengthening:

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

Postcondition weakening:

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

Example of use:

$(\!| y = 6 |\!)$

$(\!| y + 1 = 7 |\!)$ implied

```
x = y + 1
```

$(\!| x = 7 |\!)$ assignment

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.)

Proof Format: Annotated Programs

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...

$$(\!| x = x_0 \wedge y = y_0 |\!)$$

```
t = x ;
```

```
x = y ;
```

```
y = t ;
```

$$(\!| x = y_0 \wedge y = x_0 |\!)$$

Example: Composition of Assignments

To show: the following is satisfied under partial correctness.

We work bottom-up for assignments...

$$(\!| x = x_0 \wedge y = y_0 |\!)$$

```
t = x ;
```

```
x = y ;
```
$(\!| x = y_0 \wedge t = x_0 |\!)$ $\quad P_2 = (\!| P[t/y] |\!)$

```
y = t ;
```
$(\!| x = y_0 \wedge y = x_0 |\!)$ $\quad$ assignment $(\!| P |\!)$

Example: Composition of Assignments

To show: the following is satisfied under partial correctness.

We work bottom-up for assignments...

$$(\!| x = x_0 \wedge y = y_0 |\!)$$

```
t = x ;
```

$(\!| y = y_0 \wedge t = x_0 |\!)$ $\quad P_3 = (\!| P_2[y/x] |\!)$

```
x = y ;
```

$(\!| x = y_0 \wedge t = x_0 |\!)$ $\quad$ assignment

```
y = t ;
```

$(\!| x = y_0 \wedge y = x_0 |\!)$ $\quad$ assignment

Example: Composition of Assignments

To show: the following is satisfied under partial correctness.

We work bottom-up for assignments...

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

Example: Composition of Assignments

To show: the following is satisfied under partial correctness.

We work bottom-up for assignments...

$$
\begin{array}{ll}
(\!| x = x_0 \wedge y = y_0 |\!) & \\
(\!| y = y_0 \wedge x = x_0 |\!) & \text{implied } \textit{[proof required]} \\
\texttt{t = x ;} & \\
(\!| y = y_0 \wedge t = x_0 |\!) & \text{assignment} \\
\texttt{x = y ;} & \\
(\!| x = y_0 \wedge t = x_0 |\!) & \text{assignment} \\
\texttt{y = t ;} & \\
(\!| x = y_0 \wedge y = x_0 |\!) & \text{assignment}
\end{array}
$$

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

Programs with Conditional Statements

Deduction Rules for Conditionals

`if-then-else`:

$$\frac{(\!|\, P \wedge B \,|\!)\ \texttt{C}_1\ (\!|\, Q \,|\!) \qquad (\!|\, P \wedge \neg B \,|\!)\ \texttt{C}_2\ (\!|\, Q \,|\!)}{(\!|\, P \,|\!)\ \texttt{if (B) C}_1\ \texttt{else C}_2\ (\!|\, Q \,|\!)}\ \text{(if-then-else)}$$

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

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

Template for Conditionals With `else`

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

$(\!| P |\!)$
`if ( B ) {`
    $(\!| P \wedge B |\!)$ if-then-else
    C_1
    $(\!| Q |\!)$ *(justify depending on C_1—a "subproof")*
`} else {`
    $(\!| P \wedge \neg B |\!)$ if-then-else
    C_2
    $(\!| Q |\!)$ *(justify depending on C_2—a "subproof")*
`}`
$(\!| Q |\!)$ 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 $P \wedge \neg B \rightarrow Q$

Example: Conditional Code

Example: Prove the following is satisfied under partial correctness.

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

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

$(\!| \mathit{max} \geq x |\!)$

$(\!| P |\!)$

```
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.

$(\!|\,\texttt{true}\,|\!)$

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

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

```
    max = x ;
```

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

```
}
```

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

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

Doing the Steps

1. Add annotations for the `if-then` statement.
2. "Push up" for the assignments.

$(\!|\,\mathsf{true}\,|\!)$

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

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

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

```
    max = x ;
```

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

```
}
```

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

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

Doing the Steps

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

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

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

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

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

```
    max = x ;
```

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

```
}
```

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

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

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):

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

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

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

1.	$true \wedge \neg(max < x)$	assumption
2.	$\neg(max < x)$	$\wedge$e: 1
3.	$max \geq x$	def. of $\geq$
4.	$\big(true \wedge \neg(max < x)\big) \rightarrow (max \geq x)$	$\rightarrow$i: 1–3

Example 2 for Conditionals

Prove the following is satisfied under partial correctness.

$(\!|\, \mathsf{true} \,|\!)$

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

$(\!|\, (x > y \wedge max = x) \vee (x \leq y \wedge max = y) \,|\!)$

Example 2: Annotated Code

$(\!|\,\texttt{true}\,|\!)$
```
if (x > y) {
```
$(\!|\, x > y \,|\!)$ if-then-else

```
    max = x ;
```

```
} else {
```
$(\!|\, \neg(x > y) \,|\!)$ if-then-else

```
    max = y ;
```
$(\!|\, (x > y \wedge max = x) \vee (x \leq y \wedge max = y) \,|\!)$
```
}
```
$(\!|\, (x > y \wedge max = x) \vee (x \leq y \wedge max = y) \,|\!)$ if-then-else

Example 2: Annotated Code

$(\!|\,\texttt{true}\,|\!)$
```
if (x > y) {
```
$(\!|\, x > y \,|\!)$ — if-then-else

$(\!|\, (x > y \wedge x = x) \vee (x \leq y \wedge x = y) \,|\!)$
```
    max = x ;
```
$(\!|\, (x > y \wedge max = x) \vee (x \leq y \wedge max = y) \,|\!)$ — assignment
```
} else {
```
$(\!|\, \neg(x > y) \,|\!)$ — if-then-else

$(\!|\, (x > y \wedge y = x) \vee (x \leq y \wedge y = y) \,|\!)$
```
    max = y ;
```
$(\!|\, (x > y \wedge max = x) \vee (x \leq y \wedge max = y) \,|\!)$ — assignment
```
}
```
$(\!|\, (x > y \wedge max = x) \vee (x \leq y \wedge max = y) \,|\!)$ — if-then-else

Example 2: Annotated Code

$(\!\lvert \texttt{true} \rvert\!)$	
`if (x > y) {`	
$\quad(\!\lvert x > y \rvert\!)$	if-then-else
$\quad(\!\lvert (x > y \wedge x = x) \vee (x \leq y \wedge x = y) \rvert\!)$	implied (a)
`  max = x ;`	
$\quad(\!\lvert (x > y \wedge max = x) \vee (x \leq y \wedge max = y) \rvert\!)$	assignment
`} else {`	
$\quad(\!\lvert \neg(x > y) \rvert\!)$	if-then-else
$\quad(\!\lvert (x > y \wedge y = x) \vee (x \leq y \wedge y = y) \rvert\!)$	implied (b)
`  max = y ;`	
$\quad(\!\lvert (x > y \wedge max = x) \vee (x \leq y \wedge max = y) \rvert\!)$	assignment
`}`	
$(\!\lvert (x > y \wedge max = x) \vee (x \leq y \wedge max = y) \rvert\!)$	if-then-else

Example 2: Implied Conditions

(a) Prove $x > y \rightarrow (x > y \wedge x = x) \vee (x \leq y \wedge x = y)$

1.	$x > y$	assumption
2.	$x = x$	EQ1
3.	$x > y \wedge x = x$	$\wedge$i: 1, 2
4.	$(x > y \wedge x = x) \vee (x \leq y \wedge x = y)$	$\vee$i: 3
5.	$x > y \rightarrow (x > y \wedge x = x) \vee (x \leq y \wedge x = y)$	$\rightarrow$i: 1–4

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.	$x \leq y$	assumption
2.	$y = y$	EQ1
3.	$x \leq y \wedge y = y$	$\wedge$i: 1,2
4.	$(x > y \wedge y = x) \vee (x \leq y \wedge y = y)$	$\vee$i: 3
5.	$x \leq y \rightarrow \big((x > y \wedge y = x) \vee (x \leq y \wedge y = y)\big)$	$\rightarrow$i: 1–4

`While`-Loops and Total Correctness

Inference Rule: Partial-`while`

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

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

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`

$(\!| P |\!)$
$(\!| I |\!)$ Implied (a)
`while ( B ) {`
  $(\!| I \wedge B |\!)$ partial-`while`
  C
  $(\!| I |\!)$ $\longleftarrow$ *to be justified, based on* C
  }
$(\!| I \wedge \neg B |\!)$ partial-`while`
$(\!| Q |\!)$ Implied (b)

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

(b) Prove $(I \wedge \neg B) \rightarrow Q$ (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 ;
}
```

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

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

Example: Finding a loop invariant

```
    (| x ≥ 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 $(y = z!)$.

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

```
y = 1 ;

z = 0 ;
```
$(\!| y = z! |\!)$ *[justification required]*

```
while (z != x) {
```
$(\!| (y = z!) \wedge \neg(z = x) |\!)$ partial-`while` $((\!| I \wedge B |\!))$

```
    z = z + 1 ;

    y = y * z ;
```
$(\!| y = z! |\!)$ *[justification required]*

```
}
```
$(\!| y = z! \wedge z = x) |\!)$ partial-`while` $((\!| I \wedge \neg B |\!))$

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

Example: annotations for partial-`while`

Annotate assignment statements (bottom-up).

$(\!| x \geq 0 |\!)$
$(\!| 1 = 0! |\!)$
`y = 1 ;`
$(\!| y = 0! |\!)$ assignment
`z = 0 ;`
$(\!| y = z! |\!)$ assignment
`while (z != x) {`
  $(\!| (y = z!) \wedge \neg(z = x) |\!)$ partial-`while`
  $(\!| y(z+1) = (z+1)! |\!)$
  `z = z + 1 ;`
  $(\!| yz = z! |\!)$ assignment
  `y = y * z ;`
  $(\!| y = z! |\!)$ assignment
`}`
$(\!| y = z! \wedge z = x |\!)$ partial-`while`
$(\!| y = x! |\!)$

Example: annotations for partial-`while`

Note the required implied conditions.

$(\!\| x \geq 0 \|\!)$	
$(\!\| 1 = 0! \|\!)$	implied (a)
`y = 1 ;`	
$(\!\| y = 0! \|\!)$	assignment
`z = 0 ;`	
$(\!\| y = z! \|\!)$	assignment
`while (z != x) {`	
$(\!\| (y = z!) \wedge \neg(z = x) \|\!)$	partial-`while`
$(\!\| y(z+1) = (z+1)! \|\!)$	implied (b)
`z = z + 1 ;`	
$(\!\| yz = z! \|\!)$	assignment
`y = y * z ;`	
$(\!\| y = z! \|\!)$	assignment
`}`	
$(\!\| y = z! \wedge z = x) \|\!)$	partial-`while`
$(\!\| y = x! \|\!)$	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.	$(y = z!) \wedge (z = x)$	premise
2.	$y = z!$	$\wedge$e: 1
3.	$z = x$	$\wedge$e: 1
4.	$y = x!$	EqSubs: 2, 3

Example: implied condition (b)

Proof of implied (b):

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

1.	$y = z! \land z \neq x$	premise
2.	$y = z!$	$\land$e: 1
3.	$(z+1)\,y = (z+1)\,z!$	EqSubs: 2
4.	$(z+1)\,z! = (z+1)!$	def. of factorial: 3
5.	$(z+1)\,y = (z+1)!$	EqTrans: 3, 4

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 ;
}
```

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

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	$1*2*2*2$

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.

$(\!| n \geq 0 \wedge a \geq 0 |\!)$
$(\!| \ldots |\!)$
`s = 1 ;`
$(\!| \ldots |\!)$
`i = 0 ;`
$(\!| s = a^i |\!)$
`while (i < n) {`
  $(\!| s = a^i \wedge i < n |\!)$ partial-`while`
  $(\!| \ldots |\!)$
  `s = s * a ;`
  $(\!| \ldots |\!)$
  `i = i + 1 ;`
  $(\!| s = a^i |\!)$
`}`
$(\!| s = a^i \wedge i \geq n |\!)$ 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?

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

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

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

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

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

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?

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

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

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

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

$x - z$ decreases by 1

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

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 ;
                                  At start of loop: x − z = x ≥ 0
while ( z != x ) {
    z = z + 1 ;                   x − z decreases by 1
    y = y * z ;                   x − z unchanged
}
(| y = x! |)
```

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?

$(\!| x \geq 0 |\!)$
```
y = 1 ;
z = 0 ;
```
At start of loop: $x - z = x \geq 0$
```
while ( z != x ) {
    z = z + 1 ;
    y = y * z ;
}
```
$x - z$ decreases by 1

$x - z$ unchanged

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

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.

For-loops

`for`-loops — in simple form

`For v = e1 to e2 {  C  }`

- Values of `v`, e_1 and e_2 cannot be changed in C.
- If $e_1 > e_2$ then nothing is done.
- If $e_1 \leq e_2$ the `for`-statement is equivalent to

  `v = e1 ;`
  C
  `v = e1 + 1 ;`
  C
  $\vdots$
  `v = e2 ;`
  C

C is always executed $(e_2 - e_1) + 1$ times

Inference Rules for `for`-loops

Rule `for`-loop A (for $e_1 \leq e_2$):

$$\frac{(\!| I \wedge (e_1 \leq v) \wedge (v \leq e_2) |\!)\ \ C\ \ (\!| I[v+1/v] |\!)}{(\!| I[e_1/v] \wedge (e_1 \leq e_2) |\!)\ \texttt{for v = } e_1 \texttt{ to } e_2\ \{C\}\ (\!| I[e_2+1/v] |\!)}$$

Rule `for`-loop B (for $e_1 > e_2$):

$$\frac{}{(\!| I \wedge (e_1 > e_2) |\!)\ \texttt{for v = } e_1 \texttt{ to } e_2\ \{C\}\ (\!| I \wedge (e_1 > e_2) |\!)}$$

Annotation of `for`-Loops

The case $e_1 \leq e_2$: For a selected invariant I,

$(\!\| P \|\!)$	
$(\!\| I[e_1/v] \wedge (e_1 \leq e_2) \|\!)$	implied (a)
`for v = e1 to e2 {`	
$\quad (\!\| I \wedge (e_1 \leq v) \wedge (v \leq e_2) \|\!)$	`for`-loop A
$\quad C$	
$\quad (\!\| I[v+1/v] \|\!)$	[*justification required*]
`}`	
$(\!\| I[e_2+1/v] \|\!)$	`for`-loop A
$(\!\| Q \|\!)$	implied (b)

Implied (a): $P \rightarrow I[e_1/v] \wedge (e1 \leq e2)$

Implied (b): $I[e_2+1/v] \rightarrow Q$.

Annotating a `for`-Loop: Second case

The case $e_1 > e_2$: For any I,

$(\!\| P \|\!)$	
$(\!\| I \wedge (e_1 > e_2) \|\!)$	implied (a)
`for v = e1 to e2 {`	
$\quad C$	[*Nothing happens*]
`}`	
$(\!\| I \wedge (e_1 > e_2) \|\!)$	`for`-loop B
$(\!\| Q \|\!)$	implied (b)

Implied (a): $P \rightarrow I \wedge (e_1 > e_2)$

Implied (b): $I \wedge (e_1 > e_2) \rightarrow Q$.

Example of a `for`-loop

Prove the following is satisfied under partial correctness.

$$
\begin{array}{ll}
(\!| n \geq 1 |\!) & (\!| P |\!) \\
\texttt{x = 0 ;} & \\
\texttt{for i = 1 to n \{} & \\
\quad \texttt{x = x + i;} & C \\
\texttt{\}} & \\
(\!| x = n(n+1)/2 |\!) & (\!| Q |\!)
\end{array}
$$

Candidate loop invariant I: $x = i(i+1)/2$.

But it doesn't work....

`for`-Loop: Finding an invariant

Candidate invariant I: $x = i(i+1)/2$.
Loop bounds: $e_1 = 1$ and $e_2 = n$.
Post-condition Q: $x = n(n+1)/2$.

The condition at the end of the `for`-loop will be $I[e_2 + 1/i]$, which is $x = (n+1)(n+2)/2$.

The implication to get the postcondition is thus

$$\begin{array}{ll} (\!| x = (n+1)(n+2)/2 |\!) & \texttt{for}\text{-loop A} \\ (\!| x = n(n+1)/2 |\!) & \text{implied} \end{array}$$

But this implication doesn't hold!

Revised candidate for loop invariant I: $x = (i-1)i/2$.

for-Loop: Annotating the loop

I: $x = (i-1)i/2$ $\quad$ $I[i+1/i]$: $x = i(i+1)/2$

$I[1/i]$: $x = (1-1)1/2$ $\quad$ $I[n+1/i]$: $x = n(n+1)/2$

$\{\!| n \geq 1 |\!\}$

```
x = 0 ;
```
$\{\!| x = (1-1)1/2 \wedge 1 \leq n |\!\}$
```
for i = 1 to n {
```
$\quad \{\!| x = (i-1)i/2 \wedge (1 \leq i) \wedge (i \leq n) |\!\}$ $\quad$ for-loop

```
    x = x + i ;
```
$\quad \{\!| x = i(i+1)/2 |\!\}$
```
}
```
$\{\!| x = n(n+1)/2 |\!\}$ $\quad$ for-loop

$\{\!| x = n(n+1)/2 |\!\}$

for-Loop: Annotating the loop

I: $x = (i-1)i/2$ $\quad$ $I[i+1/i]$: $x = i(i+1)/2$

$I[1/i]$: $x = (1-1)1/2$ $\quad$ $I[n+1/i]$: $x = n(n+1)/2$

$\{\!| n \geq 1 |\!\}$

$\{\!| 0 = (1-1)1/2 \wedge 1 \leq n |\!\}$

```
x = 0 ;
```

$\{\!| x = (1-1)1/2 \wedge 1 \leq n |\!\}$ assignment

```
for i = 1 to n {
```

$\quad \{\!| x = (i-1)i/2 \wedge (1 \leq i) \wedge (i \leq n) |\!\}$ for-loop

$\quad \{\!| x + i = i(i+1)/2 |\!\}$

```
    x = x + i ;
```

$\quad \{\!| x = i(i+1)/2 |\!\}$ assignment

```
}
```

$\{\!| x = n(n+1)/2 |\!\}$ for-loop

$\{\!| x = n(n+1)/2 |\!\}$

for-Loop: Annotating the loop

I: $x = (i-1)i/2$ $\quad$ $I[i+1/i]$: $x = i(i+1)/2$

$I[1/i]$: $x = (1-1)1/2$ $\quad$ $I[n+1/i]$: $x = n(n+1)/2$

$(\!| n \geq 1 |\!)$

$(\!| 0 = (1-1)1/2 \wedge 1 \leq n |\!)$ implied (a)

```
x = 0 ;
```

$(\!| x = (1-1)1/2 \wedge 1 \leq n |\!)$ assignment

```
for i = 1 to n {
```

$(\!| x = (i-1)i/2 \wedge (1 \leq i) \wedge (i \leq n) |\!)$ for-loop

$(\!| x + i = i(i+1)/2 |\!)$ implied (b)

```
    x = x + i ;
```

$(\!| x = i(i+1)/2 |\!)$ assignment

```
}
```

$(\!| x = n(n+1)/2 |\!)$ for-loop

$(\!| x = n(n+1)/2 |\!)$

for-Loop: Subproofs

Implied (a): $(\!| n \geq 1 |\!) \to (\!| 0 = (1-1)1/2 \wedge 1 \leq n |\!)$

1.	$n \geq 1$	premise
2.	$0 = (1-1)1/2$	EQ2, algebra
3.	$(n \geq 1) \wedge (0 = (1-1)1/2)$	$\wedge$i: 1,2
4.	$n \geq 1 \to (n \geq 1) \wedge (0 = (1-1)1/2)$	$\to$i: 1–3

Implied (b): $x = (i-1)i/2 \wedge (1 \leq i) \wedge (i \leq n) \to x + i = i(i+1)/2.$

(Exercise.)

Arrays

Assignment of Values of an Array

Let `A` be an array of `n` integers: `A[1]`, `A[2]`, ..., `A[n]`.

Assignment may work as before:

$$(\!| P[A[x]/v] |\!)$$
`v = A[x] ;`
$$(\!| P |\!)$$
assignment

But a complication can occur:

$$(\!| A[y] = 0 |\!)$$
`A[x] = 1;`
$$(\!| A[y] = 0 |\!)$$???

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.

Assignment to a Whole Array

Our solution: Treat an assignment to an array value

$$\texttt{A[}e_1\texttt{]} \texttt{ = } e_2 \texttt{ ;}$$

as an assignment of the whole array

$$\texttt{A = A\{}e_1 \leftarrow e_2\texttt{\}} \texttt{ ;}$$

where the term "$\texttt{A\{}e_1 \leftarrow e_2\texttt{\}}$" denotes an array identical to A except the e_1^{th} element is changed to have the value e_2.

Array Assignment: Definition and Examples

Definition:

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

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:

$$\frac{}{(\!|\, P[A\{e_1 \leftarrow e_2\}/A] \,|\!)\ \texttt{A[e}_1\texttt{]} \texttt{ = e}_2\ (\!|\, P \,|\!)}\text{(Array assignment)}$$

where

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

Example

Prove the following is satisfied under partial correctness.

$$(\!| A[x] = x_0 \wedge A[y] = y_0 |\!)$$

```
t = A[x] ;
```

```
A[x] = A[y] ;
```

```
A[y] = t ;
```

$$(\!| A[x] = y_0 \wedge A[y] = x_0 |\!)$$

We do assignments bottom-up, as always. . . .

Example: push up conditions for assignments

$(\!| A[x] = x_0 \wedge A[y] = y_0 |\!)$

```
t = A[x] ;
```

```
A[x] = A[y] ;
```

$(\!| A\{y \leftarrow t\}[x] = y_0 \wedge A\{y \leftarrow t\}[y] = x_0 |\!)$

```
A[y] = t ;
```

$(\!| A[x] = y_0 \wedge A[y] = x_0 |\!)$

array assignment

Example: push up conditions for assignments

$$(\!| A[x] = x_0 \wedge A[y] = y_0 |\!)$$

```
t = A[x] ;
```

$$(\!| A\{x \leftarrow A[y]\}\{y \leftarrow t\}[x] = y_0 \wedge A\{x \leftarrow A[y]\}\{y \leftarrow t\}[y] = x_0 |\!)$$

```
A[x] = A[y] ;
```

$$(\!| A\{y \leftarrow t\}[x] = y_0 \wedge A\{y \leftarrow t\}[y] = x_0 |\!)$$ array assignment

```
A[y] = t ;
```

$$(\!| A[x] = y_0 \wedge A[y] = x_0 |\!)$$ array assignment

Example: push up conditions for assignments

$$(\!| A[x] = x_0 \wedge A[y] = y_0 |\!)$$

$$(\!| A\{x \leftarrow A[y]\}\{y \leftarrow A[x]\}[x] = y_0 \wedge A\{x \leftarrow A[y]\}\{y \leftarrow A[x]\}[y] = x_0 |\!)$$

```
t = A[x] ;
```

$$(\!| A\{x \leftarrow A[y]\}\{y \leftarrow t\}[x] = y_0 \wedge A\{x \leftarrow A[y]\}\{y \leftarrow t\}[y] = x_0 |\!)$$

assignment

```
A[x] = A[y] ;
```

$$(\!| A\{y \leftarrow t\}[x] = y_0 \wedge A\{y \leftarrow t\}[y] = x_0 |\!)$$

array assignment

```
A[y] = t ;
```

$$(\!| A[x] = y_0 \wedge A[y] = x_0 |\!)$$

array assignment

Example: push up conditions for assignments

$(\!| A[x] = x_0 \wedge A[y] = y_0 |\!)$

$(\!| A\{x \leftarrow A[y]\}\{y \leftarrow A[x]\}[x] = y_0 \wedge A\{x \leftarrow A[y]\}\{y \leftarrow A[x]\}[y] = x_0 |\!)$ — implied (a)

```
t = A[x] ;
```

$(\!| A\{x \leftarrow A[y]\}\{y \leftarrow t\}[x] = y_0 \wedge A\{x \leftarrow A[y]\}\{y \leftarrow t\}[y] = x_0 |\!)$ — assignment

```
A[x] = A[y] ;
```

$(\!| A\{y \leftarrow t\}[x] = y_0 \wedge A\{y \leftarrow t\}[y] = x_0 |\!)$ — array assignment

```
A[y] = t ;
```

$(\!| A[x] = y_0 \wedge A[y] = x_0 |\!)$ — 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

$$A\{x \leftarrow A[y]\}\{y \leftarrow A[x]\}[y] = A[x] \quad .$$

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, reverse the elements.

Algorithm: exchange $R[j]$ with $R[n+1-j]$, for each j.

A possible program is

```
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 post-condition, and a loop invariant.

Reversal code: conditions and an invariant

Pre-condition: $\forall x \cdot \big((1 \le x \le n) \to (R[x] = r_x)\big)$.

Post-condition: $\forall x \cdot \big((1 \le x \le n) \to (R[x] = r_{n+1-x})\big)$.

Invariant? When exchanging at position j?

> If $x < j$, then $R[x]$ and $R[n+1-x]$ have already been exchanged.
>
> If $x \ge j$, $x \le n+1-j$, then no exchange has happened yet.

Thus let $Inv(j)$ be the formula

$$\forall x \cdot \Big(\big(1 \le x < j \to (R[x] = r_{n+1-x} \wedge R[n+1-x] = r_x)\big) \\ \wedge \big(j \le x \le n/2 \to (R[x] = r_x \wedge R[n+1-x] = r_{n+1-x})\big)\Big) \ .$$

Reversal: annotations around the loop

The annotations surrounding the `while`-loop:

$(\!\| n \geq 0 \wedge \forall x \cdot \big((1 \leq x \leq n) \rightarrow (R[x] = r_x)\big) \|\!)$	
$(\!\| \mathit{Inv}(1) \|\!)$	implied (a)
`j = 1 ;`	
$(\!\| \mathit{Inv}(j) \|\!)$	assignment
`while ( 2*j <= n ) {`	
$(\!\| \mathit{Inv}(j) \wedge 2j \leq n \|\!)$	partial-`while`
$\vdots$	
$(\!\| \mathit{Inv}(j) \|\!)$	*(TBA)*
`}`	
$(\!\| \mathit{Inv}(j) \wedge 2j > n \|\!)$	partial-`while`
$(\!\| \forall x \cdot \big((1 \leq x \leq n) \rightarrow (R[x] = r_{n+1-x})\big) \|\!)$	implied (b)

Reversal code: annotations inside the loop

We must now handle the code inside the loop.

$(\!| \mathit{Inv}(j) \wedge 2j \leq n |\!)$ partial-`while`

$(\!| \mathit{Inv}(j+1)[R'/R]$, *where* R' *is*
$R\{j \leftarrow R[n+1-j]\}\{(n+1-j) \leftarrow R[j]\} |\!)$ implied (c)

```
t = R[j]; R[j] = R[n+1-j]; R[n+1-j] = t;
```

$(\!| \mathit{Inv}(j+1) |\!)$ Lemma

```
j = j + 1 ;
```

$(\!| \mathit{Inv}(j) |\!)$ assignment

The Implied Condition

Recall $\mathit{Inv}(j)$:

$$\forall x \cdot \Big(\big(1 \le x < j \to (R[x] = r_{n+1-x} \land R[n+1-x] = r_x) \big) \\ \land \big(j \le x \le n/2 \to (R[x] = r_x \land R[n+1-x] = r_{n+1-x}) \big) \Big) \ .$$

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

$$\forall x \cdot \Big(\big(1 \le x < j+1 \to (R'[x] = r_{n+1-x} \land R'[n+1-x] = r_x) \big) \\ \land \big(j+1 \le x \le n/2 \to (R'[x] = r_x \land R'[n+1-x] = r_{n+1-x}) \big) \Big) \ ,$$

which by the construction of R' is equivalent to

$$\forall x \cdot \Big(\big(1 \le x < j \to (R[x] = r_{n+1-x} \land R[n+1-x] = r_x) \big) \\ \land R'[j] = r_{n+1-j} \land R'[n+1-j] = r_j \\ \land \big(j+1 \le x \le n/2 \to (R[x] = r_x \land R[n+1-x] = r_{n+1-x}) \big) \Big) \ .$$

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.

Precondion: A is sorted: $\forall i \cdot \forall j \cdot \big((1 \leq i < j \leq n) \rightarrow (A[i] \leq A[j])\big)$.

Output values: boolean `found`; integer m.

Post-condition: Either `found` is true and $A[m] = x$, or `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: Outer Loop

$(\!| \forall i \cdot \forall j \cdot \big((1 \le i < j \le n) \to (A[i] \le A[j])\big) |\!)$

```
l = 1;    u = n;    found = false;
```

$(\!| I |\!)$

```
while ( l <= u and !found ) {
```

$(\!| I \wedge (l \le u \wedge \neg \texttt{found}) |\!)$ partial-while

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

$(\!| J |\!)$

```
    if (A[m] = x) {
```

. . . Body omitted. . .

```
    }
```

$(\!| Q |\!)$ if-then-else

$(\!| I |\!)$

```
}
```

$(\!| I \wedge \neg(l \le u \wedge \neg \texttt{found}) |\!)$ partial-while

$(\!| (found \wedge A[m] = x) \vee (\neg found \wedge \forall k \cdot \neg(A[k] \neq x)) |\!)$

Code: Inner "if"

$(\!| J |\!)$

```
if ( A[m] = x ) {
```

$(\!| J \wedge (A[m] = x) |\!)$ if-then-else

```
    found = true;
```

$(\!| Q |\!)$

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

$(\!| J \wedge \neg(A[m] = x) \wedge (A[m] < x) |\!)$ if-then-else

```
    l = m+1;
```

$(\!| Q |\!)$

```
} else {
```

$(\!| J \wedge \neg(A[m] = x) \wedge \neg(A[m] < x) |\!)$ if-then-else

```
    u = m - 1;
```

$(\!| Q |\!)$

```
}
```

$(\!| Q |\!)$ if-then-else

An Extended Example: Sorting

Post-Condition for Sorting

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

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

First Attempt

$(\!| n \geq 1 |\!)$

$C_{\texttt{sort}}$

$(\!| \mathit{sorted}(A, n) |\!)$

Post-Condition for Sorting

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

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

First Attempt

$(\!| n \geq 1 |\!)$

$C_{\texttt{sort}}$

$(\!| \mathit{sorted}(A, n) |\!)$

$(\!| n \geq 1 |\!)$

```
for i = 1 to n {
    A[i] = 0 ;
}
```

$(\!| \mathit{sorted}(A, n) |\!)$

Post-Condition for Sorting, II

Let *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

$(\!| n \geq 1 \wedge A = A' |\!)$

$C_{\texttt{sort}}$

$(\!| \texttt{sorted}(A, n) \wedge$
$\texttt{permutation}(A, A', n) |\!)$

Post-Condition for Sorting, II

Let *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

$(\!| n \geq 1 \wedge A = A' |\!)$

$C_{\texttt{sort}}$

$(\!| \texttt{sorted}(A, n) \wedge \texttt{permutation}(A, A', n) |\!)$

$(\!| n \geq 1 \wedge A = A' |\!)$

`n = 1 ;`
some algorithm on A ;

$(\!|$ *sorted*$(A, n) \wedge$ *permutation*$(A, A', n) |\!)$

Post-Condition for Sorting, III

Final Attempt (Correct)

$$(\!| n \geq 1 \wedge n = n_0 \wedge A = A' |\!)$$

$$C_{\texttt{sort}}$$

$$(\!| \mathit{sorted}(A, n_0) \wedge \mathit{permutation}(A, A', n_0) |\!)$$

Algorithms for Sorting

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

Input: Array A, with indices $A[1] \ldots A[n]$.

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
> > Insert $A[k]$ into its correct place among $A[1] \ldots A[k-1]$:
> >
> > > Compare it with $A[k-1]$, $A[k-2]$, etc., until its proper place is reached.

Insertion Sort: Inserting one element

Possible code for the insertion loop:

```
i = k ;
while ( i > 1 ) {
    if ( A[i] < A[i-1] ) {
        t = A[i] ;
        A[i] = A[i-1] ;
        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 A 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. When finding a large element appearing before a small element, exchange them.

Requisite: Do all exchanges in a single scan. (Linear time, not quadratic!)

(A diagram of the algorithm, and its code, do not fit well on an overhead. See separate notes on Learn.)