Program Verification
Outline

• Introduction: What and Why?
• Pre- and Post-conditions
• Conditionals
• while-Loops and Total Correctness
• for-Loops
• Arrays
• Termination (total correctness)
• 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 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 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
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 $\Phi_R$ 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.
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
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 \( y \).

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

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

State at the “while” test:

- Initial state \(s_0\): \(x=0, z=0, y=1\)
- 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

$y = 1$;
$z = 0$;

$\rightarrow$ while (z != x) {
    $z = z + 1$;
    $y = y \times z$;
}

State at the “while” test:

- Initial state $s_0$: $x=0$, $z=0$, $y=1$
- 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!
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 $x = -4$?
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 $x$. 
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.
Hoare Triples

Our assertions about programs will have the form

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

- \( \langle P \rangle \) — precondition
- \( C \) — program or code
- \( \langle Q \rangle \) — 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: $\langle P \rangle\ C\ \langle Q \rangle$.

**Example.** The requirement

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

might be expressed as

$$\langle x > 0 \rangle\ C\ \langle y \times y < x \rangle.$$
Specification Is Not Behaviour

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

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

\[
C_1:
\begin{align*}
y & = 0 ; \\
\text{while} \ (y \times y < x) & \{
\begin{align*}
y & = y + 1 ; \\
\end{align*}
\}
\end{align*}
\]

\[
C_2:
\begin{align*}
y & = 0 ; \\
\text{while} \ (y \times y < x) & \{
\begin{align*}
y & = y + 1 ; \\
\end{align*}
\}
\end{align*}
\]

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 Natural Deduction for Predicate Logic, 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 $\langle P \rangle C \langle Q \rangle$ is satisfied under partial correctness, denoted

$$\vdash_{\text{par}} \langle P \rangle C \langle Q \rangle,$$

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

For example, the program

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

These are called **logical variables**.

**Example.**

\[
\{ \, x = x_0 \land x_0 \geq 0 \, \} \\
y = 1; \\
while \ (x \neq 0) \{ \\
\hspace{1em} y = y \times x; \\
\hspace{1em} x = x - 1; \\
\} \\
\{ \, y = x_0! \, \}
\]
Logical variables

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

These are called **logical variables**.

**Example.**

\[
\begin{align*}
\{ & \ x = x_0 \land x_0 \geq 0 \ \} \\
& y = 1; \\
\text{while} \ (x \neq 0) \ {\{} \\
& \quad y = y \ast x; \\
& \quad x = x - 1; \\
{\} \\
\{ & \ y = x_0! \ \}
\end{align*}
\]

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 (to be defined later, before For-Loops) 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.

\[
\begin{align*}
\{ \text{program precondition} \} \\
y = 1; \\
\{ \text{... } \} \\
\{ \text{while} \ (x \neq 0) \} \\
\{ \text{... } \} \\
y = y \times x; \\
\{ \text{... } \} \\
x = x - 1; \\
\{ \text{... } \} \\
\{ \text{program postcondition} \} \\
\end{align*}
\]
Inference Rule for Assignment

\[ \frac{Q[E/x]}{x = E \parallel Q} \]  

(assignment)

Intuition:

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

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

by one application of the assignment rule.
More Examples for Assignment

Example 1.

(\( y = 2 \))  \( \langle Q[E/x] \rangle \)
\( x = y ; \)  \( x = E; \)
(\( x = 2 \))  \( \langle Q \rangle \)

Example 2.

(\( 0 < 2 \))  \( \langle Q[E/x] \rangle \)
\( x = 2 ; \)  \( x = E; \)
(\( 0 < x \))  \( \langle Q \rangle \)
Examples of Assignment

Example 3.

\[
\begin{align*}
(x + 1) = 2 \\ x = x + 1 \\
(x = 2)
\end{align*}
\]

Example 4.

\[
\begin{align*}
(x + 1) = n + 1 \\
x = x + 1; \\
(x = n + 1)
\end{align*}
\]
Note about Examples

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

\[
\begin{align*}
???
& \quad \langle Q[E/x] \rangle \\
x = y; \quad x = E; \\
\langle x > 0 \rangle & \quad \langle Q \rangle
\end{align*}
\]
Inference Rules with Implications

Rule of “Precondition strengthening”:

\[
\frac{P \rightarrow P'}{\langle P \rangle \ C \ \langle Q \rangle} \quad \text{(implied)}
\]

Rule of “Postcondition weakening”:

\[
\frac{\langle P \rangle \ C \ \langle Q' \rangle \quad Q' \rightarrow Q}{\langle P \rangle \ C \ \langle Q \rangle} \quad \text{(implied)}
\]

Example of use:

\[
\begin{align*}
\langle y = 6 \rangle \\
\langle y + 1 = 7 \rangle & \quad \text{implied} \\
x &= y + 1 \\
\langle x = 7 \rangle & \quad \text{assignment}
\end{align*}
\]
Inference Rule for Sequences of Instructions

\[
\frac{\langle P \rangle C_1 \langle Q \rangle, \langle Q \rangle C_2 \langle R \rangle}{\langle P \rangle C_1; C_2 \langle R \rangle} \quad \text{(composition)}
\]

In order to prove \(\langle P \rangle C_1; C_2 \langle R \rangle\), 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.)
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.
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...

\[
\begin{align*}
\{ x = x_0 \land y = y_0 \} \\
t = x ; \\
x = y ; \\
y = t ; \\
\{ x = y_0 \land y = x_0 \}
\end{align*}
\]
Example: Composition of Assignments

To show: the following is satisfied under partial correctness.

We work bottom-up for assignments...

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

\( t = x ; \)

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

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

\( P_2 \) is \( \{ P[t/y] \} \)

assignment \( \{ P \} \)
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...

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

To show: the following is satisfied under partial correctness.

We work bottom-up for assignments...

\[
\begin{align*}
( x = x_0 \land y = y_0 ) \\
( y = y_0 \land x = x_0 ) \\
t = x ; \\
( y = y_0 \land t = x_0 ) \\
x = y ; \\
( x = y_0 \land t = x_0 ) \\
y = t ; \\
( x = y_0 \land y = x_0 )
\end{align*}
\]

implied \([proof \ required]\)

Finally, show

\[
( x = x_0 \land y = y_0 ) \implies ( y = y_0 \land x = x_0 ) .
\]

This is clear.
Programs with Conditional Statements
Deduction Rules for Conditionals

if-then-else:

\[
\frac{\langle P \land B \rangle \quad C_1 \quad \langle Q \rangle \quad \langle P \land \neg B \rangle \quad C_2 \quad \langle Q \rangle}{\langle P \rangle \quad \text{if (B)} \quad C_1 \quad \text{else} \quad C_2 \quad \langle Q \rangle} \quad \text{(if-then-else)}
\]

if-then (without else):

\[
\frac{\langle P \land B \rangle \quad C \quad \langle Q \rangle \quad (P \land \neg B) \rightarrow Q}{\langle P \rangle \quad \text{if (B)} \quad C \quad \langle Q \rangle} \quad \text{(if-then)}
\]
Template for Conditionals With else

Annotated program template for if-then-else:

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

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

\[
(Q) \quad \text{ if-then} \\
\text{Implied: Proof of } P \land \neg B \rightarrow Q
\]
Example: Prove the following is satisfied under partial correctness.

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

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 "implieds":
   • Annotations from (1) above containing implications
   • Adjacent assertions created in step (2).

Proofs here are written in Math 135 style. They can use Predicate Logic, basic arithmetic, or any other appropriate reasoning.
1. **Add annotations** for the if–then statement.

```plaintext
(true)
if ( max < x ) {
  (true ∧ max < x)  // if-then
  max = x ;
  (max ≥ x) ← to be shown
}

(max ≥ x)  // if-then
Implied: (true ∧ ¬(max < x)) → max ≥ x
```
Doing the Steps

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

\[
\text{\texttt{if ( max < x ) \{}}
\]
\[
\text{\texttt{\quad ( true \land max < x ) \land x \geq x \land max \geq x \land true \land \neg \texttt{max < x} \rightarrow max \geq x }}
\]
Doing the Steps

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

\[
\begin{align*}
(\text{true}) \\
\text{if ( max < x )} \\
\hspace{1em} (\text{true} \land \text{max < x}) & \quad \text{if-then} \\
\hspace{1em} (x \geq x) & \quad \text{Implied (a)} \\
\text{max = x ;} \\
\hspace{1em} (\text{max} \geq x) & \quad \text{assignment} \\
\end{align*}
\]

\[
(\text{max} \geq x) \quad \text{if-then} \\
\text{Implied (b): } (\text{true} \land \neg (\text{max < x})) \rightarrow \text{max} \geq x
\]
The auxiliary “implied” proofs can be done in Math 135 style (and assuming the necessary arithmetic properties). We will write them informally, but clearly.

Proof of Implied (a):

\[ \emptyset \vdash (true \land (max < x)) \rightarrow (x \geq x) \].

- Clearly \((x \geq x)\) is a tautology and so the required implication holds.
Implied (b)

Proof of Implied (b): Show \( \emptyset \vdash ((P \land (\neg B)) \rightarrow Q) \), which in this case is

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

- The hypothesis, \((\text{true} \land (\neg (\text{max} < x)))\) can be simplified to \((\neg (\text{max} < x))\).
- Then by properties of < and \(\geq\), the conclusion, \((\text{max} \geq x)\), follows.
Example 2 for Conditionals

Prove the following is satisfied under partial correctness.

\[
\{ \text{true} \} \\
\text{if ( } x > y \text{ ) \{} \\
\quad \text{max} = x; \\
\text{\} else \{} \\
\quad \text{max} = y; \\
\text{\}} \\
\{ (x > y \land max = x) \lor (x \leq y \land max = y) \}
\]
Example 2: Annotated Code

```

|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) |
```

Example 2: Annotated Code

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

\[
\begin{align*}
\&true & \\
\text{if} & \ ( \ x \ > \ y \ ) \ \{ & \\
\&x \ > \ y & \ \text{if-then-else} \\
\& ( x \ > \ y \ \& \ x = x ) \ \lor \ ( x \ \leq \ y \ \& \ x = y ) & \ \text{implied (a)} \\
\text{max} & = x ; \\
\& ( x \ > \ y \ \& \ \text{max} = x ) \ \lor \ ( x \ \leq \ y \ \& \ \text{max} = y ) & \ \text{assignment} \\
\} & \ \text{else} \ \{ & \\
\&\neg ( x > y ) & \ \text{if-then-else} \\
\& ( x \ > \ y \ \& \ y = x ) \ \lor \ ( x \ \leq \ y \ \& \ y = y ) & \ \text{implied (b)} \\
\text{max} & = y ; \\
\& ( x > y \ \& \ \text{max} = x ) \ \lor \ ( x \ \leq \ y \ \& \ \text{max} = y ) & \ \text{assignment} \\
\} \\
\& ( x > y \ \& \ \text{max} = x ) \ \lor \ ( x \ \leq \ y \ \& \ \text{max} = y ) & \ \text{if-then-else}
\end{align*}
\]
Example 2: Implied Conditions

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

- From \((x > y)\) and from the trivially true \((x = x)\), it follows that \(((x > y) \land (x = x))\).
- From this it follows that \(((x > y) \land (x = x)) \lor ((x \leq y) \land (x = y)))\).

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

- From \((x \leq y)\) and from the trivially true \((y = y)\), it follows that \(((x > y) \land (y = x))\).
- From this it follows that \(((x > y) \land (y = x)) \lor ((x \leq y) \land (y = y)))\).
While-Loops and Total Correctness
A triple $⟦P⟧ C ⟦Q⟧$ is satisfied under total correctness, denoted

$$\models_{\text{tot}} ⟦P⟧ C ⟦Q⟧,$$

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:

\[
\begin{align*}
\{ x = 1 \} \\
\{ y = x \} \\
\{ y = 1 \}
\end{align*}
\]

Example 2. Neither total nor partial correctness:

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

\[ \{ x = 1 \} \]
while (true) {
    \[ \{ x = 0 \} \]
} \[ \{ x > 0 \} \]
Example 4. Total correctness

\( x \geq 0 \)

\( y = 1 ; \)
\( z = 0 ; \)

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

\( y = x! \)

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

\[ x \geq 0 \]
\[ y = 1 ; \]
while (x != 0) {
    \[ y = y \times x ; \]
    \[ x = x - 1 ; \]
} 

\[ y = x! \]
Inference Rule: Partial-while

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

\[
\begin{align*}
&\langle I \land B \rangle \ C \ (I) \\
\Rightarrow &
\langle I \rangle \ \text{while} \ (B) \ C \ \langle I \land \neg B \rangle
\end{align*}
\]

(partial-while)

In words:
If the code \( C \) satisfies the triple \( \langle I \land B \rangle \ C \ (I) \), 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 \textit{loop invariant}.

After the while-loop terminates, \( \neg B \) is also true.
Annotations for Partial-while

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

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

(b) Prove \( (I \land \neg B) \rightarrow Q \) (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

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

\( y = 1 \)
\( z = 0 \)

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

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

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

At the `while` statement:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></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>
Example: Finding a loop invariant

\((x \geq 0)\)

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

\begin{align*}
\text{At the while statement:} \\
x & \quad y & \quad z & \quad z \neq x \\
5 & \quad 1 & \quad 0 & \quad \text{true} \\
5 & \quad 1 & \quad 1 & \quad \text{true} \\
5 & \quad 2 & \quad 2 & \quad \text{true} \\
5 & \quad 6 & \quad 3 & \quad \text{true} \\
5 & \quad 24 & \quad 4 & \quad \text{true} \\
5 & \quad 120 & \quad 5 & \quad \text{false}
\end{align*}

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 \((y = z!)\).

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

Annotate assignment statements (bottom-up).

\[
\begin{align*}
&\{ x \geq 0 \} \\
&\{ 1 = 0! \} \\
&y = 1 \\
&\{ y = 0! \} \quad \text{assignment} \\
&z = 0 \\
&\{ y = z! \} \quad \text{assignment} \\
&\text{while } (z \neq x) \{ \\
&\quad \{ (y = z!) \land \neg(z = x) \} \\
&\quad \{ y(z + 1) = (z + 1)! \} \\
&\quad z = z + 1 \\
&\quad \{ yz = z! \} \quad \text{assignment} \\
&\quad y = y \times z \\
&\quad \{ y = z! \} \quad \text{assignment} \\
&\} \\
&\{ y = z! \land z = x \} \quad \text{partial-while} \\
&\{ y = x! \}
\end{align*}
\]
Example: annotations for partial-while

Note the required implied conditions.

\[ \{ x \geq 0 \} \]
\[ \{ 1 = 0! \} \]
\[ y = 1 ; \]
\[ \{ y = 0! \} \]
\[ z = 0 ; \]
\[ \{ y = z! \} \]

while (z != x) {

\[ \{ (y = z!) \land \neg(z = x) \} \]
\[ \{ y(z + 1) = (z + 1)! \} \]
\[ z = z + 1 ; \]
\[ \{ yz = z! \} \]
\[ y = y * z ; \]
\[ \{ y = z! \} \]

}

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

implied (a)
assignment
implied (b)
assignment
assignment
implied (c)
Example: implied conditions (a) and (c)

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

This result follows by the definition of factorial.

Proof of implied (c): \( \{((y = z!) \land (z = x))\} \vdash (y = x!)\).

From \( z = x \), we obtain \( z! = x! \). Then since \( y = z! \), by the transitivity of equality, we obtain \( y = z! = x! \).
Proof of implied (b):

\[ \left\{ \left( y = z! \right) \land \neg\left( z = x \right) \right\} \vdash (z + 1) y = (z + 1)! . \]

If \( y = z! \), then we can multiply both sides by \( (z + 1) \), to obtain
\[ y(z + 1) = (z + 1)z! = (z + 1)! . \]
Example 2 (Partial-while)

Prove the following is satisfied under partial correctness.

\[ (n \geq 0 \land a \geq 0) \]

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

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

Prove the following is satisfied under partial correctness.

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

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

s = a^n
```

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.

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

Trace of the loop:

\[
\begin{array}{cccc}
\text{a} & \text{n} & \text{i} & \text{s} \\
2 & 3 & 0 & 1 \\
2 & 3 & 1 & 1 \times 2 \\
2 & 3 & 2 & 1 \times 2 \times 2 \\
2 & 3 & 3 & 1 \times 2 \times 2 \times 2 \\
\end{array}
\]

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.
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.

\[
\begin{align*}
\langle n \geq 0 \land a \geq 0 \rangle \\
\langle 1 = a^0 \land 0 \leq n \rangle \\
s = 1 \\
\langle s = a^0 \land 0 \leq n \rangle \\
i = 0 \\
\langle s = a^i \land i \leq n \rangle \\
\text{while } (i < n) \{ \\
\langle s = a^i \land i \leq n \land i < n \rangle \\
\langle s \cdot a = a^{i+1} \land i + 1 \leq n \rangle \\
s = s \cdot a \\
\langle s = a^{i+1} \land i + 1 \leq n \rangle \\
i = i + 1 \\
\langle s = a^i \land i \leq n \rangle \\
\} \\
\langle s = a^i \land i \leq n \land i \geq n \rangle \\
\langle s = a^n \rangle 
\end{align*}
\]

implied (a)

assignment

assignment

partial-while

implied (b)

assignment

assignment

partial-while

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?

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

\[
\text{while ( z != x ) {}
\begin{align*}
  &z = z + 1 ; \\
  &y = y * z ; \\
\}
\]

\[
\text{( y = x! )}
\]
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?

\[
\begin{align*}
\text{At start of loop: } & x - z = x \geq 0 \\
\text{while } \ (z \neq x) \{ & \text{ } \\
& z = z + 1 ; \quad x - z \text{ decreases by 1} \\
& y = y \times z ; \\
\} \\
\text{ } & y = x! \}
\end{align*}
\]
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 ;
}
```
```
\[ y = x! \]
```

$x - z$ decreases by 1

$x - z$ unchanged

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?

\[
\langle x \geq 0 \rangle
\]
\[
y = 1 ;
\]
\[
z = 0 ;
\]

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

while ( $z \neq x$ ) {
    \[
z = z + 1 ;
    \]
    $x - z$ decreases by 1
    \[
y = y \ast z ;
    \]
    $x - z$ unchanged
}

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

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 *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 $z \neq x$ ends the loop.
Arrays
Assignment of Values of an Array


Assignment may work as before: 

\[
\langle P[A[x]/v] \rangle \\
\text{v} = A[x] ; \\
\langle P \rangle \quad \text{assignment}
\]

But a complication can occur: 

\[
\langle A[y] = 0 \rangle \\
A[x] = 1; \\
\langle A[y] = 0 \rangle \quad ???
\]

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

A correct rule must account for possible changes to $A[y], A[z+3], \text{etc.},$ 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 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:

\[
\langle Q[A\{e_1 \leftarrow e_2\}/A]\rangle \ A[e_1] = e_2 \ \langle Q \rangle
\]

(Array assignment)

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.

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

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

\( \langle A[x] = x_0 \land A[y] = y_0 \rangle \)

t = A[x] ;


\( \langle A\{y \leftarrow t\}[x] = y_0 \land A\{y \leftarrow t\}[y] = x_0 \rangle \)

A[y] = t ;

\( \langle A[x] = y_0 \land A[y] = x_0 \rangle \)

array assignment
Example: push up assertions for assignments

\[
\langle A[x] = x_0 \land A[y] = y_0 \rangle
\]

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

A[y] = t ;
\[
\langle A[x] = y_0 \land A[y] = x_0 \rangle
\]
array assignment
Example: push up assertions for assignments

\[
\begin{align*}
\langle A[x] = x_0 \land A[y] = y_0 \rangle \\
\langle A\{x \leftarrow A[y]\}\{y \leftarrow A[x]\}\}[x] = y_0 \\
\quad \land A\{x \leftarrow A[y]\}\{y \leftarrow A[x]\}\}[y] = x_0 \rangle
\end{align*}
\]

\[
t = A[x] ;
\]

\[
\begin{align*}
\langle A\{x \leftarrow A[y]\}\{y \leftarrow t\}\}[x] = y_0 \\
\quad \land A\{x \leftarrow A[y]\}\{y \leftarrow t\}\}[y] = x_0 \rangle
\end{align*}
\]

\[
\]

\[
\begin{align*}
\langle A\{y \leftarrow t\}\}[x] = y_0 \land A\{y \leftarrow t\}\}[y] = x_0 \rangle
\end{align*}
\]

\[
A[y] = t ;
\]

\[
\langle A[x] = y_0 \land A[y] = x_0 \rangle
\]
Example: push up assertions for assignments

\[
\begin{align*}
\langle & A[x] = x_0 \land A[y] = y_0 \rangle \\
\langle & A\{x \leftarrow A[y]\}\{y \leftarrow A[x]\}[x] = y_0 \\
& \quad \land A\{x \leftarrow A[y]\}\{y \leftarrow A[x]\}[y] = x_0 \rangle \\
\end{align*}
\]

impeded (a)

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

assignment

\[
\begin{align*}
\langle & A\{y \leftarrow t\}[x] = y_0 \land A\{y \leftarrow t\}[y] = x_0 \rangle \\
A[y] &= t \\
\langle & A[x] = y_0 \land A[y] = x_0 \rangle \\
\end{align*}
\]

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

```plaintext
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: \( (\forall x \left((1 \leq x \leq n) \rightarrow (R[x] = r_x)\right)) \).

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

Invariant? When has an exchange occurred at position \( x \)?

- 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

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

and \( Inv(j) = Inv'(j) \land (1 \leq j \leq n/2+1) \).
The annotations surrounding the \texttt{while}-loop:

\[
\begin{align*}
&\{(\n \geq 0) \land (\forall x ((1 \leq x \leq n) \rightarrow (R[x] = r_x)))\}\} \implies \text{Implied (a)} \\
&\{\text{Inv}(1)\} \\
&j = 1 ; \\
&\{\text{Inv}(j)\} \implies \text{Assignment} \\
&\text{while ( } 2*j \leq n ) \{ \\
&\quad \{ (\text{Inv}(j) \land (2j \leq n))\} \implies \text{Partial-While} \\
&\quad \vdots \\
&\quad \{\text{Inv}(j)\} \\
&\} \\
&\{ (\text{Inv}(j) \land (2j > n))\} \implies \text{Partial-While} \\
&\{ (\forall x ((1 \leq x \leq n) \rightarrow (R[x] = r_{n+1-x}))\} \implies \text{Implied (b)}
\end{align*}
\]
We must now handle the code inside the loop.

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

partial-while
implied (c)
Lemma
assignment
Proof of Implied Condition (c)

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

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

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

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

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

$$
\left( \forall x \left( ((1 \leq x < j) \rightarrow (R[x] = r_{n+1-x} \land R[n+1-x] = r_x))
\land (R'[j] = r_{n+1-j}) \land (R'[n+1-j] = r_j)
\land ((j + 1 \leq x \leq n/2) \rightarrow (R[x] = r_x \land R[n+1-x] = r_{n+1-x})) \right) \right).
$$
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: $\forall i \forall j ((1 \leq i < j \leq n) \rightarrow (A[i] \leq A[j]))$.

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

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

\[ l = 1; \quad u = n; \quad found = false; \]

\[ I \]

while \( l \leq u \text{ and } !found \) {
  \[ I \land (l \leq u \land \neg found) \]
  \[ J \]
  if \( A[m] = x \) {
    ...
  }
  \[ I \]
}

\[ I \land \neg(l \leq u \land \neg found) \]

\[ (found \land A[m] = x) \lor (\neg found \land \forall k \neg(A[k] = x)) \]
Code: the if-statement

\[ J \]
if ( A[m] = x ) {
  \[ J \land (A[m] = x) \] if-then-else
  found = true;
\[ I \]
} else if ( A[m] < x ) {
  \[ J \land \neg(A[m] = x) \land (A[m] < x) \] if-then-else
  l = m+1;
\[ I \]
} else {
  \[ J \land \neg(A[m] = x) \land \neg(A[m] < x) \] if-then-else
  u = m - 1;
\[ I \]
}
\[ I \] if-then-else

Program Verification

Example: Binary Search
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.)
\( \forall i \; \forall j \; (1 \leq i < j \leq n) \rightarrow (A[i] \leq A[j]) \) \}

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

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

while ( \( l <= u \) && !\text{found} ) \{

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

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

while

\( (\forall i \; ((A[i] = x) \rightarrow (l \leq i \leq u))) \land \neg \text{found} \land (l \leq \lfloor (l + u)/2 \rfloor \leq u) \) \}

implied

m = (l+u) div 2 ;

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

assignment

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

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

if ( A[m] = x ) {

\[ (true \to A[m] = x) \land (\forall i ((A[i] = x) \to (\ell \leq i \leq u))) \]

implied

found = true;

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

assignment

}

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

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

if ( A[m] < x ) {
    \[
    \forall i \left( (A[i] = x) \rightarrow (\ell \leq i \leq u) \right) \land \neg \text{found} \land (\ell \leq m \leq u) \land \neg (A[m] = x) \land (A[m] < x)
    \]
    \[
    (\text{found} \rightarrow A[m] = x) \land \left( \forall i \left( (A[i] = x) \rightarrow (m + 1 \leq i \leq u) \right) \right)
    \]
    l = m + 1;
    \[
    (\text{found} \rightarrow A[m] = x) \land \left( \forall i \left( (A[i] = x) \rightarrow (\ell \leq i \leq u) \right) \right)
    \]
}

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
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 $A[1] \leq A[2] \leq ... \leq A[n]$.

**First Attempt**

\[
\{ n \geq 1 \} \quad C_{\text{sort}} \quad \{ \text{sorted}(A, n) \}
\]
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

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

\[
\begin{align*}
\{ & n \geq 1 \land A = A' \\
\} & \text{sort}
\end{align*}
\]

\[ C_{\text{sort}} \]

\[
\{ \text{sorted}(A, n) \land \text{permutation}(A, A', n) \} \]
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**

\[
\{ n \geq 1 \land A = A' \} \quad \{ n \geq 1 \land A = A' \}
\]

\[
C_{\text{sort}} \quad \text{some algorithm on } A ; \\
\{ sorted(A, n) \land n = 1 ; \\
\{ sorted(A, n) \land \text{permutation}(A, A', n) \} \quad \{ sorted(A, n) \land \text{permutation}(A, A', n) \}
\]
Postcondition for Sorting, III

Final Attempt (Correct)

\[
\{ n \geq 1 \land n = n_0 \land A = A' \}
\]

\[ C_{\text{sort}} \]

\[
\{ \text{sorted}(A, n_0) \land \text{permutation}(A, A', n_0) \}
\]
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


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 $A[k - 1], A[k - 2], \text{etc.}$, 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-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.

![Partition: The Algorithm](image)

One step of the algorithm:

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

Program Verification
Extended Example: Sorting
Code and Pre- and Postconditions

\[
\begin{align*}
\langle \ n &\geq 1 \ \rangle \\
a &= 1 \\
\text{while ( } a < n \ \&\& \ X[a] \leq p \ \text{) } \{ \\
\quad a &= a + 1 \\
\}
\]

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

\[
\langle \ \exists z \left( (1 \leq z \leq n + 1) \ \&\ (X[1..z] \leq p) \ \&\ (X[z..n] > p) \right) \ \rangle
\]

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:

$$\begin{align*}
\{ (X[1..a] \leq p) \land ((a \geq n) \lor (X[a] > p)) \} .
\end{align*}$$

Annotation for the while, and pushing up, yields

\[
\begin{align*}
\{ (X[1..1] \leq p) \} & \quad \text{implied} \\
a = 1 ; & \\
\{ (X[1..a] \leq p) \} & \quad \text{assignment} \\
\text{while} \ (a < n \land X[a] \leq p) \{ & \\
\{ (X[1..a] \leq p) \land ((a < n) \land (X[a] \leq p)) \} & \quad \text{partial-while implied} \\
\{ (X[1..a+1] \leq p) \} & \\
a = a + 1 ; & \\
\{ (X[1..a] \leq p) \} & \quad \text{assignment} \\
\}
\end{align*}
\]

$$\begin{align*}
\{ (X[1..a] \leq p) \land ((a \geq n) \lor (X[a] > p)) \} .
\end{align*}$$
The Second \texttt{while}-Loop

For the second \texttt{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
Inside the while-Loop

“And” the loop guard to the invariant at the start of the loop. Then pushing up through the assignment and if yields

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

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

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

Push up for the assignments inside the if:

\[ (X[1..a] \leq p) \land (X[a..b] > p) \land (b \leq n) \land X[b] \leq p \]
if-then

\[ ((X[1..a] \leq p)) \land (X[a] > p) \land (X[a+1..b] > p) \land X[b] \leq p \]
if-then

implied

\[ t = X[b] \land X[b] = X[a] \land X[a] = t \land ((X[1..a] \leq p)) \land (X[a] \leq p) \land (X[a+1..b] > p) \land X[b] > p \]
if-then

implied

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

implied

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

(The extra “implied” just makes the “swap” clearer.)
The annotation thus far works fine. But there is a “glitch”….

Between the loops, we have

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

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

\[
\begin{align*}
(1 \leq a \leq n) \land (X[1..a) \leq p) \land ((a = n) \lor (X[a..a + 1) > p)) & \quad \text{partial-while} \\
(X[1..a) \leq p) \land ((a = n) \lor (X[a..a + 1) > p)) & \quad \text{implied}
\end{align*}
\]

\[
b = a + 1 ;
\]

\[
(1 \leq a \leq n) \land ((a = n) \lor (X[a..b) > p)) & \quad \text{assignment}
\]

while ( b <= n ) {

\[
(1 \leq a \leq n) \land ((a = n) \lor (X[a..b) > p)) \land (b \leq n) & \quad \text{partial-while}
\]

\[
(X[1..a) \leq p) \land (X[a..b) > p) \land (b \leq n) & \quad \text{implied}
\]

if (X[b] \leq p ) {

\[
(X[1..a) \leq p) \land (X[a..b) > p) & \quad \text{implied}
\]

if-then

}\}

\[
(1 \leq a \leq n) \land ((a = n) \lor (X[a..b) > p)) & \quad \text{implied}
\]

}\}

\[
(1 \leq a \leq n) \land ((a = n) \lor (X[a..b) > p)) \land (b > n) & \quad \text{partial-while}
\]

\[
\exists z ((1 \leq z \leq n + 1) \land (X[1..z) \leq p) \land (X[z..n] > p)) & \quad \text{implied}
\]

“Implied” proofs left to you.
CS245 — Logic and Computation

Undecidability and the Halting Problem

Jonathan Buss

with thanks to

Borzoo Bonakdarpour, Daniela Maftuleac and Tyrel Russell
What is Computability?

From an informal or intuitive perspective what might we mean by computability?

One natural interpretation is that something is **computable** if it can be calculated by a systematic procedure.

One might think that, given enough resources and a sufficiently sophisticated program, a computer could solve any problem.

However, some problems cannot be automated.
What is Computability?

An instruction such as “guess the correct answer” does not seem to be systematic.

An instruction such as “try all possible answers” is less clear cut: it depends on whether the possible answers are finite or infinite in number.
A **decision problem** is a problem which calls for an answer of either **yes** or **no**, on some input.

**Examples**

1. Given a formula of propositional logic, is it satisfiable?
2. Given a positive integer, is it prime?
3. Given a graph and two of its vertices, is there a path between the two vertices?
4. Given a multivariate polynomial with integer coefficients, does it have any integer roots?
5. Given a program and input, will the program terminate on the input?
Sometimes an apparently reasonable decision problem has subtleties.

**Example: The Barber Paradox**

There is a barber who is said to shave all men, and only those men, who do not shave themselves. Who shaves the barber?

As phrased, the barber could be a woman. But what if we insist the barber is a man?

Then if the barber shaves himself it is because he does not shave himself, and in turn this is because he does shave himself.

“Does the barber shave himself?” is unanswerable.
A Curious Problem

Do the following terminate, given a positive integer $n$?

```scheme
while ( n > 1 ) {
    if ( n is even ) {
        n = n/2 ;
    } else {
        n = 3*n + 1 ;
    }
}
```

```
(define (C n)
    (cond
        ((<= n 1 ) 1 )
        ((even? n) (C (/ n 2) ) )
        (else ( C (+ (* 3 n) 1) ) )
    )
)
```

Nobody knows!

Nevertheless, it is a decision problem.
A decision problem is *decidable* if there is an algorithm that, given an input to the problem,

- outputs “yes” (or “true”) if the input has answer “yes” and
- outputs “no” (or “false”) if the input has answer “no”.

A problem is *undecidable* if it is not decidable.
One of the best-known undecidable problems is the *Halting Problem*. Given a program $P$ and an input $I$, will $P$ terminate if run on input $I$?

E.g.: A Scheme program that does not terminate:

```
(define (loop x) (loop loop))
```

Can we write a program which takes as an input any program $P$ and input $I$ for $P$ and returns true, if the program terminates (halts) on $I$ and returns false, otherwise?
Testing Whether a Program Halts

;; Contract: halts?: SchemeProgram Input → boolean
;;
;; If the evaluation of (P I) halts, then (halts? P I) halts with value true, and
;; If the evaluation of (P I) does not halt, then
;; (halts? P I) halts with the value false.
;;
;; Example: (halts? loop loop) returns false,
;; given definition (define (loop x) (loop loop))

(define (halts? P I)

)
Theorem.

No Scheme function can perform the task required of \texttt{halts?}, correctly for all programs.

That is, the Halting Problem is undecidable.
**Proof**: By contradiction.

Assume that there exists some function \((\text{halts?} \ P \ I)\) which returns \(true\) if the program \(P\) halts on input \(I\) and returns \(false\) if \(P\) does not halt on input \(I\).

A key idea: one possible input to the program \(P\) would be the program \(P\) itself. Therefore, \((\text{halts?} \ P \ P)\) would return \(true\) if the program halts when given itself as an input.

A function \text{halts?} could be called by other functions.
Let’s consider the following function, which uses \texttt{halts}?.

\begin{verbatim}
(define ( self-halt? P )
  ( halts? P P ))
\end{verbatim}

This should determine whether $P$ terminates when given itself as input.

What happens if we call (\texttt{self-halt? self-halt?})?
“What Do I Do?”

\[
\text{(self-halt? self-halt?)}
\]

\[
\Rightarrow \text{(halts? (self-halt? self-halt?))}
\]

\[
\Rightarrow ...
\]

\[
\Rightarrow \begin{cases} 
\text{true,} & \text{if (self-halt? self-halt?) halts} \\
\text{false,} & \text{if (self-halt? self-halt?) does not halt.}
\end{cases}
\]

Since evaluation of `halts?` always terminates, the evaluation of 
`(self-halt? self-halt?)` also terminates. And, since `halts?` gives the 
correct answer, the final result of the execution must be true.
OK, now let’s consider the following function.

```scheme
(define (halt-if-loop P)
  (cond
   [(halts? P P) (loop loop)]
   [else true])
)
```

What happens if we invoke `halt-if-loop` with itself as its argument?
“Aacckk!”

\[
\begin{align*}
\text{(halt-if-loop &amp; halt-if-loop)} &amp; \Rightarrow \text{(cond [ ( halts? (halt-if-loop &amp; halt-if-loop) )}
\notag
\text{ (loop loop)]}
\notag
\text{[ else true ]}
\notag
\) &amp; \Rightarrow \ldots
\notag
\Rightarrow \left\{ \begin{array}{l}
\text{(loop loop),}
\notag
\text{if (halt-if-loop &amp; halt-if-loop) halts,}
\notag
\text{true, if (halt-if-loop &amp; halt-if-loop) does not halt.}
\end{array} \right.
\end{align*}
\]

The evaluation of (halt-if-loop &amp; halt-if-loop) terminates iff the evaluation of (halt-if-loop &amp; halt-if-loop) does not terminate.

Program halt-if-loop cannot exist!
Thus also program halts? cannot exist.
The proof of the undecidability of the halting problem uses a technique called **diagonalization**, devised by Georg Cantor in 1873.

Cantor was concerned with the problem of measuring the sizes of infinite sets. For two infinite sets, how can we tell whether one is larger than the other or whether they are of the same size?

**Example.** The set of even natural numbers is the same size (!!) as the set of all natural numbers.

**Example.** The set of all infinite sequences over \( \{0, 1\} \) is larger than the set of natural numbers.
The “diagonalization” we did for the halting problem is rather tricky. To prove some other problem undecidable, it helps to have another method.

We shall describe the method known as reduction between problems.

First, we note that you already know the method, even though you may not know its name....
Reduction Between Problems

A *reduction* from problem A to problem B is an algorithm (or program) to solve problem A that relies on an algorithm (or program) to solve B.

Example:

*Problem A:* given an array, find its median element. (That is, the element s.t. half of the elements are smaller and half are larger.)

*Problem B:* given an array, sort it.

An algorithm for problem A:

• Sort the array (i.e., solve problem B).

• Look at position \( n/2 \), where \( n \) is the length of the array.

The above is a reduction from A to B.
The Significance of a Reduction

Suppose we have a reduction from A to B.

• If we can solve B, then we can solve A.
  Even if we didn’t write the code for B — maybe it’s in a “library” — we can still use it.
  This is a very common and useful technique.

• Conversely, if A is undecidable, then B is undecidable.
  No one can write code for A, since it’s undecidable.
  Thus no one can write code for B, either.
  It would provide code for A, but such code doesn’t exist.
Undecidability Via Reduction

Let problem A be the Halting Problem:
Given \((P \; I)\), does \(P\) halt on input \(I\)?

Let problem B be the “Looping Problem”:
Given \((P \; I)\), does \(P\) run forever on input \(I\)?

Suppose we had an algorithm \(\text{loops}\)\? for the Looping Problem. Then we could write the following:

\[
(\text{define} \ (\text{halts}\? \ P \ I) \\
(\text{cond} \ [(\text{loops}\? \ P \ I) \ #f] \\
\quad [\text{else} \ #t])
\)

If the function \(\text{loops}\)\? solves the Looping Problem, then the function \(\text{halts}\)\? solves the Halting Problem. Since the latter is impossible, the former is also impossible.
Another Undecidable Problem

Provability of a formula in FOL:
Given a formula \( \varphi \), does \( \varphi \) have a proof?
(Or, equivalently, is \( \varphi \) valid?)

No algorithm exists to solve provability:

**Theorem** Provability is undecidable.
Outline of proof: Suppose that some algorithm $Q$ decides provability.

1. Devise an algorithm to solve the following problem:
   Given a program $(P, I)$, produce a formula $\varphi_{P,I}$ such that $\varphi_{P,I}$ has a proof iff $(P, I)$ halts.
   (Base the formula on Step and Eval.)

2. Combine algorithm $Q$ with the above algorithm to get an algorithm that decides the Halting Problem: On input $(P, I)$, give the formula $\varphi_{P,I}$ as input to $Q$.

But no algorithm decides the Halting problem.
Thus no algorithm decides provability.
The Post Correspondence Problem (devised by Emil Post)

Given a finite sequence of pairs \((s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)\) such that all \(s_i\) and \(t_i\) are binary strings of positive length, is there a sequence of indices \(i_1, i_2, \ldots, i_n\) with \(n \geq 1\) such that the concatenation of the strings \(s_{i_1} s_{i_2} \ldots s_{i_n}\) equals \(t_{i_1} t_{i_2} \ldots t_{i_n}\)?
An Instance of the Post Correspondence Problem

Suppose we have the following pairs: (1,101), (10,00), (011,11). Can we find a solution for this input?

Yes. Indices (1,3,2,3) work: $s_1 s_3 s_2 s_3$ equals $t_1 t_3 t_2 t_3$, as both yield 101110011.

What about the pairs (001,0), (01,011), (01,101), (10,001)?

In this case, there is no sequence of indices.

Remember that an index can be used arbitrarily many times. This gives us some indication that the problem might be unsolvable in general, as the search space is infinite.
The problem

Given a multivariate polynomial equation, does it have any integer solutions? is undecidable.

The proof is similar in principle: show that deciding whether an equation has integral solutions allows deciding whether a program halts.

The details, however, get quite complicated.

The issue was publicly raised in 1900 (by Hilbert) and not solved until 1971 (by Matjasevič and Robinson).