Part 5: Program Verification

Contents [DOCUMENT FINALIZED]

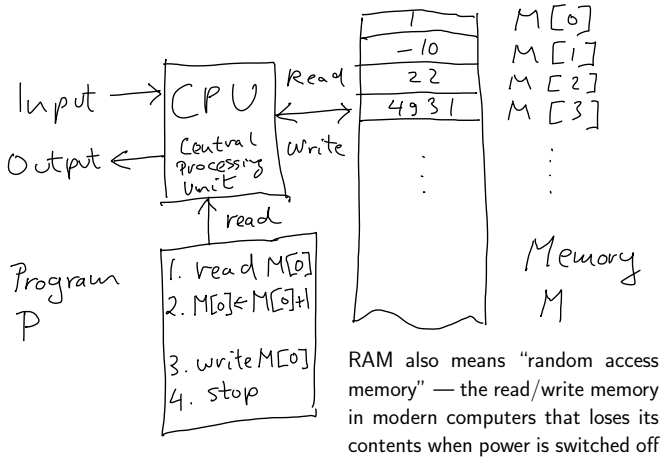
Computing Functions

In this last lecture part we will first introduce a computation model that allows us to specify simple numerical algorithms.

Then we will formalize the concept of program correctness, develop a formal method to prove statements about computations, and then apply it to proving the correctness of simple algorithms.

Computation Model

RAM (Random Access Machine)



RAM also means "random access memory" — the read/write memory in modern computers that loses its contents when power is switched off

- Input/Output devices: function arguments read from input, computed result written to output
- CPU (Central Processing Unit): executes program
- Memory M: infinite sequence of integers that form the variables that can be used by the program
- Program P: fixed, finite, instruction sequence the CPU executes step by step

CPU

- has direct access to any memory location (by index)
- moves integers between memory cells
- reads integers from input to memory
- writes integers from memory to output
- executes primitive operations
- contains a program counter (p) that is advanced by one after each step or set to a specific value by jump instructions

The RAM Instruction Set Overview

RAM program building blocks:

- input/output instructions
- copy memory contents
- (conditional) jumps
- arithmetic operations ($+, -, *$, div, mod)
- comparisons
- stop — ends execution

Example 1:
RAM Program that computes $f(x) = x + 1$ for $x \in \mathbb{Z}$

```
1. read M[0]          // read number x into M[0]
2. M[0] ← M[0]+1      // add 1 to M[0] (now x+1)
3. write M[0]         // report result
4. stop               // done
```

(// indicates a comment)

The execution of this program proceeds as follows:

- Program counter (p) initialized with 1
- CPU reads first program instruction and executes it: reads a number from the input and stores it into M[0]. After, p is incremented to 2
- CPU reads second program instruction and executes it: reads M[0], adds 1, and stores result back into M[0]. After, p is incremented to 3
- CPU reads third instruction and executes it: writes M[0] to the output device, p incremented to 4
- CPU reads instruction 4 and executes it: the program stops

Example 2:
RAM Program that computes $f(x) = |x|$, the absolute value of $x \in \mathbb{Z}$, i.e.

$$|x| = \begin{cases} x, & \text{if } x \geq 0 \\ -x, & \text{if } x < 0 \end{cases}$$

```
1. read M[0]            // read number x into M[0]
2. if M[0] ≥ 0 goto 4   // proceed at 4 if x ≥ 0
3. M[0] ← - M[0]        // negate x
4. write M[0]           // report result
5. stop                 // done
```

Again, the program is executed step by step starting with instruction 1.

In line 2 we see a conditional jump statement, that checks a Boolean condition (M[0] ≥ 0) and continues the computation at instruction 4 if the result is true, or resumes with the following statement (3) if the result is false.

So, when the execution reaches instruction 4, M[0] contains the absolute value of input x: x if input x was ≥ 0, and $-x$ if input $x < 0$.

Example 3:
RAM Program that computes $f : \mathbb{N} \to \mathbb{N}$, with $f(n) = \sum_{i=1}^{n} i$.

```
// assume input ≥ 0
1. read M[0]              // read number n into i (M[0])
2. M[1] ← 0               // initialize partial sum M[1]
3. if M[0] ≤ 0 goto 7     // exit loop when i ≤ 0
4. M[1] ← M[1] + M[0]     // add i to partial sum
5. M[0] ← M[0] - 1        // decrease i
6. goto 3                 // iterate
7. write M[1]             // report result
8. stop                   // done
```

In addition to the conditional jump instruction in line 3 there is an unconditional jump instruction in line 6, whose effect is simply to set p to 3.

Lines 3–6 form a so-called while-loop which is executed repeatedly and exited when a certain condition holds. Here, the loop stops and the program execution continues at line 7 when M[0] becomes ≤ 0.

This program computes the sum of all natural numbers $1..n$ by counting down i from n to 0 and adding i to a variable that contains the partial sum up to that point.

Formalizing Program Execution

Lecture 23

What does program execution mean exactly?

Each step changes the state of the machine which can be thought of a computation snapshot.

Given a state, the computation that follows is uniquely determined.

For our RAM model, the state is given by

- the program counter p,
- the current memory contents M, which is an infinite integer sequence,
- the infinite input sequence I and input index i that marks the position of the next integer to be read,
- the infinite output sequence O and output index o that marks the position where the next output is stored

In short, a RAM state is given by a tuple

$$(M, I, O, p, i, o)$$

Using the concept of states, we can precisely define what single RAM instructions do: they change the current state into a new state.

I.e. each instruction r defines a state transition function given by

$$t_r(M, I, O, p, i, o) = (M', I', O', p', i', o')$$

E.g., for instruction 'M[0] $\leftarrow$ 1', the corresponding image (i.e. next state) is:

M'	$= M$	except for $M'[0]$ which is 1
I'	$= I$	no input changes
O'	$= O$	no output changes
p'	$= p + 1$	program counter incremented
i'	$= i$	no input changes
o'	$= o$	no output changes

For instruction 'write M[1]', the next state is:

M'	$= M$	no memory changes
I'	$= I$	no input changes
O'	$= O$	except for $O'[o] = M[1]$
p'	$= p + 1$	program counter incremented
i'	$= i$	no input changes
o'	$= o + 1$	output location incremented

What is the state when the computation starts?

M	$= (0, 0, \ldots)$	memory cleared
I	$= (i_1, \ldots, i_n, 0, \ldots)$	some input values
O	$= (0, 0, \ldots)$	output cleared
p	$= 1$	index of first instruction
i	$= 0$	first input location
o	$= 0$	first output location

How do we tell the program has stopped?

We could define $p = 0$ to indicate the program reached a stop instruction, i.e.

$$t_{\text{stop}}(M, I, O, p, i, o) = (M, I, O, 0, i, o)$$

So, now what does computation mean?

A computation is a sequence of state transitions that beginning with a start state defined above that encodes some function arguments, applies transition functions given by the current program instruction (program line p) until an end state is reached (e.g. $p = 0$). The result of the computation, i.e. the image of the arguments, is stored in $O[0..o-1]$.

Computation trace of program:

```
1. read M[0]            // read number x into M[0]
2. if M[0] ≥ 0 goto 4   // proceed at 4 if x ≥ 0
3. M[0] ← - M[0]        // negate x
4. write M[0]           // report result
5. stop                 // done
```

on input -8:

Step	M	I	O	p	i	o	Instr.
1	$(0, 0, \ldots)$	$(-8, 0, \ldots)$	$(0, 0, \ldots)$	1	0	0	read
2	$(-8, 0, \ldots)$	$(-8, 0, \ldots)$	$(0, 0, \ldots)$	2	1	0	if
3	$(-8, 0, \ldots)$	$(-8, 0, \ldots)$	$(0, 0, \ldots)$	3	1	0	M[0] $\leftarrow$
4	$(8, 0, \ldots)$	$(-8, 0, \ldots)$	$(0, 0, \ldots)$	4	1	0	write
5	$(8, 0, \ldots)$	$(-8, 0, \ldots)$	$(8, 0, \ldots)$	5	1	1	stop
6	$(8, 0, \ldots)$	$(-8, 0, \ldots)$	$(8, 0, \ldots)$	0	1	1	

output

Pseudo-Code

Modern computers use the von Neumann computer architecture which resembles RAMs. The main differences are:

- memory M is finite and can only store small numbers (say from 0 to 255) in each memory cell,
- program input and output are stored in M,
- and so are programs themselves.

In the early days of computing, programs looked like RAM instruction sequences. They were hard to understand and maintain.

Nowadays, we can still write programs that way, using so called assembly programming languages, because the underlying hardware principles haven't changed much.

However, most programming today is done using high-level programming languages such as C++, Java, C#, or Lisp.

For our purposes (and later in CMPUT 204/304) the preferred way of presenting algorithms is in pseudo-code

notation, which describes instruction sequences using English phrases and variable names, and flow control statements which are easier to understand than RAM instructions.

At the same time, pseudo-code avoids the burden of syntactic strictness which is present in all high-level languages.

Here are pseudo-code representations of the RAM programs we encountered so far:

```
// input:  integer x
// output:  x + 1
function plus1(x)
return x + 1
```

```
// input:  integer x
// output:  |x|
function abs(x)
if x < 0 then
   x ← −x
end
return x
```

```
// input:  integer n ≥ 0
// output:  ∑_{i=1}^{n} i
function sum(n)
s ← 0
while n > 0 do
   s ← s + n
   n ← n − 1
end
return s
```

Note, how the input and output mechanism in RAMs

is replaced by explicit function arguments and return statements, which allows us to compose functions quite easily — like so:

$$x \leftarrow \max(f(a), g(b) + 1, h(a, b))$$

To evaluate this expression, first function f is called with argument a, then function g is called with b and 1 is added to the result, and then h is called with arguments a, b.

In the end, these three values are passed on as arguments to the function called $\max$ and the result is stored in variable x.

Common pseudo-code statements:

```
a ← a + 1          // simple expression and assignment

while a < b do     // while-loop
   ...             // loop body executed as long as cond. true
end

for i ← a to b do  // for-loop
   ...             // loop body executed with i = a, a + 1, ..b
end                // body not executed if a > b

if a > b then      // if-then-else
   ...             // executed when condition true
else               // optional else-branch
   ...             // executed when condition false
end

x ← max(3, 4, 5)   // function call

return value       // return function value to caller
```


Variables reside in distinct memory locations. Their indexes are irrelevant.

Also, in pseudo-code programs we don't need to bother with instruction addresses anymore, because explicit goto instructions are replaced by flow control statements such as `if-then-else` and `while`.

Arrays

To study algorithms that act on arbitrarily large data sets, we will allow integer arrays of the form

$$A[1..n]$$

representing a sequence of n integers stored in memory, starting with index 1.

As usual, array elements can be read and written to:

```
A[i] ← 0 // store 0 at the i-th location
x ← A[1] // copy the first element into x
```

Conceptually, arrays are functions from an index set, like $\{1..n\}$ in the above example, to $\mathbb{Z}$.

To have RAMs support arrays, we need to add instruc-

tions with indirect addressing, e.g.

$$M[0] \leftarrow M[M[1]]$$

which when executed first reads $M[1]$, say a, and then copies $M[a]$ into $M[0]$. This way, arbitrary memory locations can be accessed.

Pseudo-code example with array:

```
// input:  integer array A[1..n]
// output:  maximum element in A[1..n]

function max(A[1..n])
m ← A[1]
for i ← 2 to n do
   if A[i] > m then
      m ← A[i]
   end
end
return m
```

Is this function correct?

In what follows, we will formalize this question and study techniques to verify program correctness.

Program Correctness

We call a program (or algorithm) correct if it meets its input-output specification.

Partial correctness simply requires that if an answer is returned it will be correct.

(Total) correctness in addition requires that the program terminates.

Thus, correctness proofs of programs that are meant to compute a certain function require two distinct steps:

1. We need to prove that the given program terminates for all inputs in the function domain, and
2. we need to show that, if the program stops, it returns the correct output value.

The general problem of proving correctness of arbitrary programs is impossible to solve algorithmically with RAMs, as the following theorem shows which we state without proof:

Theorem: The halting problem for RAM programs is undecidable, i.e. there is no RAM program that takes the code of an arbitrary RAM program P and its input x as input and reports 1 if P started on x halts, and 0 otherwise. □

But this doesn't mean that we can't establish the correctness of particular programs we are interested in.

Example:

We are looking for a program P' that given variable $n \in \mathbb{N}$ sets variable z to n^n, i.e. for input 0 the output is 1, for input 2 it is $2 \cdot 2 = 4$, and for 4 it is $3 \cdot 3 \cdot 3 = 27$.

We write this functional specification as follows:

$\{\ n \geq 0\ \}$ // precondition in { }
P'
$\{\ p = n^n\ \}$ // postcondition in { }

where the assertions in curly braces are predicates ranging over the program variables. The first assertion is assumed to be true, and all subsequent ones need to be proved true.

There is no procedure that can generate programs from such specifications automatically.

So, programmers need to rely on their experience and knowledge of definitions and program design patterns to find suitable implementations.

Consider for instance the following pseudo-code instruction sequence P':

```
i ← n              // counter
p ← 1              // partial product
while i > 0 do     // enter loop if i > 0   (*)
   i ← i − 1       // decrement counter
   p ← p ∗ n       // update partial product
end
```

Execution trace for $n = 3$ (state snapshots taken at line $(*)$):

t	n	i	p
1	3	3	1
2	3	2	3
3	3	1	9
4	3	0	27

After 3 iterations (at the fourth time line $(*)$ is visited), i has reached 0, the `while`-loop stops, and p contains $27 = 3^3$, which is correct.

For the proof that P' is correct **for all inputs** n, we will insert further comments in curly brackets that describe the relationships between program variables.


```
{ n ≥ 0 }
i ← n                                  // counter
p ← 1                                  // partial product
{ p = n^(n−i) ∧ i ≥ 0 }_1              // loop invariant
while i > 0 do                         // enter loop if i > 0   (*)
   { p = n^(n−i) ∧ i > 0 }_2
   i ← i − 1                           // decrement counter
   { p = n^(n−i−1) ∧ i ≥ 0 }_3
   p ← p ∗ n                           // update partial product
   { p = n^(n−i) ∧ i ≥ 0 }_4           // loop invariant holds again
end
{ p = n^(n−i) ∧ i ≥ 0 ∧ i ≤ 0 }_5      // loop inv.+loop exit cond.
{ p = n^(n−i) ∧ i = 0 }_6
{ p = n^n }
```

Assertion justifications:

1: $(n \geq 0 \ \wedge \ i = n) \ \Rightarrow \ (n^{n-i} = n^0 = 1 = p \ \wedge \ i \geq 0)$

2: while-condition $i > 0$ is true if the execution gets here and $p = n^{n-i}$ holds at first and after each loop iteration.

3: $i \geq 0$ now because $i > 0$ before i was decremented, $p = n^{n-i}$ before i was decremented, so now $p = n^{n-(i+1)} = n^{n-i-1}$

4: $p = n^{n-i-1}$ before $p \leftarrow p * n$, so now $p = n^{n-i}$

5: while-condition false: $i \leq 0$ and $p = n^{n-i} \ \wedge \ i \geq 0$

6: $(i \geq 0 \ \wedge \ i \leq 0) \ \Rightarrow \ i = 0$

So, if program P' exits the while-loop, variable p contains value n^n for any input $n \geq 0$, i.e. we proved the partial correctness of program P'.

For this we have used a so-called **loop invariant**, which is a predicate that is true before entering a loop, and stays true after executing the loop body.

What is left is to show that P' terminates for all inputs $n \geq 0$, i.e. for any $n \geq 0$, the while-loop is exited eventually.

For this we consider a suitable **runtime bound** that is always ≥ 0 and is decreased by each loop iteration if the while-condition is true.

In our case we can use one of the variables — i — as runtime bound. At the start, $i \geq 0$ and i is decremented by one in each loop iteration if $i > 0$.

Therefore, the number of loop iterations is finite. To see this consider the initial value of i: n. After exactly n iterations, i reaches 0 and the loop is exited.

This shows that P' terminates for each input value $n \geq 0$, and P' is (totally) correct.

Lecture 24

Formalisation

To treat the subject of program verification formally we need to specify the syntax and semantics of the programming language and the assertion language we will be using.

The **syntax** of a language defines which symbol sequences are valid, whereas its **semantics** specifies the meaning of valid symbol sequences.

With these in place, we will then discuss a formal system, called Hoare calculus, for reasoning about partial program correctness, present a systematic way to proving while-loops correct, and finally apply it to several examples.

For the purpose of this high-level introduction we will discuss concepts on a "naive" level to avoid getting bogged down in details.

Programming Language

We will be using pseudo-code with integer variables and integer arrays as described earlier.

The program semantics is defined by the state transition functions corresponding to statements and value semantics of arithmetic and Boolean expressions (i.e. how to evaluate expressions given variable values and constants).

Assertion Language

For assertions we will use predicate logic statements with $\mathbb{Z}$ as quantifier domain and program variables as the only free variables (i.e. not bound by quantifiers).

For specifying predicates we will allow functions and relations over $\mathbb{Z}$, like so

$$\forall i\ [(0 \le i < n) \Rightarrow (A[i] < x)]$$

Here, x, n, and array A are program variables, i is not!

Substitution

Let φ be a predicate logic sentence (also referred to as **formula**) in which x is a free variable, i.e. x is not in the scope of a $\exists x$ or $\forall x$ quantifier.

Example:

$$\varphi = (\exists z : z + z = x) \qquad (\text{“}x \text{ is even”})$$

Let t be an arithmetic expression over the program variables, which we call **term**.

Then $\varphi(t/x)$

[read “phi t for x”] is created from φ by replacing each free occurence of x in φ by term t.

The goal of this syntactic substitution process is for $\varphi(t/x)$ to mean for t what φ meant for x.

Example using φ above:

$$\varphi(y/x) = (\exists z : z + z = y) \qquad (\text{“}y \text{ is even”})$$

One needs to be careful to avoid conflicts with quantified variables: E.g., $\varphi(z/x)$ can't possibly mean

$$(\exists z : z + z = z)$$

The latter formula is always true, independent of the value of variable z in the program execution, whereas φ meant that x is even.

We can solve this problem by renaming all clashing quantified variables **before** substituting t. E.g.

$$\varphi(z/x) = \exists z' : z' + z' = z$$

With this groundwork we are ready to introduce program correctness statements which are named after Charles A. R. Hoare, who is best known for discovering QuickSort.

Hoare Triples

Definition:

Syntax:

For program P and formulas

$$\varphi(x_1, \ldots, x_m) \text{ and } \psi(x_1, \ldots, x_m)$$

(with free program variables $x_1, \ldots, x_m$) we call

$$\{\varphi\}\ P\ \{\psi\}$$

Hoare triple.

Semantics:

A **Hoare triple is valid** iff the following statement holds for all possible states:

If φ is true before P is executed and P stops, then ψ is true after executing P.

Thus, Hoare triples can be used to model partial program correctness.

Examples

$$\{n \geq 0\} \;\; P' \;\; \{p = n^n\} \;\; \text{is } \textbf{valid}$$

$$\{x = 7\} \;\; x \leftarrow x + 1 \;\; \{x = 8\} \;\; \text{is } \textbf{valid}$$

$$\{x \leq 7\} \;\; x \leftarrow x + 1 \;\; \{x \leq 8\} \;\; \text{is } \textbf{valid}$$

$$\{x < 0\} \;\; x \leftarrow x + 1 \;\; \{x > 0\} \;\; \text{is } \textbf{invalid}$$

$\{x > 0\}$ `while` $x > 0$ `do` $x \leftarrow x + 1$ `end` $\{x \leq 0\}$

is **valid**, because the loop exits only if $x \leq 0$ holds, nevermind the loop never exiting!

Hoare Calculus

We now consider a formal inference rule system — the **Hoare calculus** — which

- helps us carrying out formal correctness proofs,
- provides insights into how to create while-loops for specific purposes,
- has become the basis of computer aided or even computer generated correctness proofs, and
- allows us to identify theoretical limitations of program verification.

Hoare Rules

A **Hoare rule** has the form

$$\frac{H_1, H_2, \ldots, H_n}{H}$$

where H is a Hoare triple and the H_i are either Hoare triples or predicate logic formulas.

A Hoare rule is **valid** iff

$$H_1 \wedge H_2 \wedge \cdots \wedge H_n \Rightarrow H$$

is a tautology.

Example: Composition Rule

Given two programs P_1, P_2 and their matching post- and pre-conditions, the following **composition rule** asserts that after executing P_1 followed by P_2 the post-condition of P_2 holds, if before the pre-condition of P_1 was true and both programs stop:

$$\frac{\{\varphi\}\ P_1\ \{\psi\}\ ,\ \{\psi\}\ P_2\ \{\chi\}}{\{\varphi\}\ P_1\ P_2\ \{\chi\}}$$

Hoare Rule Set

Composition Rule:
$$\frac{\{\varphi\}\ P_1\ \{\psi\}\ ,\ \{\psi\}\ P_2\ \{\chi\}}{\{\varphi\}\ P_1\ P_2\ \{\chi\}}$$

If-Then Rule:
$$\frac{\{\varphi \wedge \beta\}\ P\ \{\psi\}\ ,\ (\varphi \wedge \neg\beta) \Rightarrow \psi}{\{\varphi\}\ \texttt{if}\ \beta\ \texttt{then}\ P\ \texttt{end}\ \{\psi\}}$$

If-Then-Else Rule:
$$\frac{\{\varphi \wedge \beta\}\ P_1\ \{\psi\}\ ,\ \{\varphi \wedge \neg\beta\}\ P_2\ \{\psi\}}{\{\varphi\}\ \texttt{if}\ \beta\ \texttt{then}\ P_1\ \texttt{else}\ P_2\ \texttt{end}\ \{\psi\}}$$

While Rule:
$$\frac{\{\varphi \wedge \beta\}\ P\ \{\varphi\}}{\{\varphi\}\ \texttt{while}\ \beta\ \texttt{do}\ P\ \texttt{end}\ \{\varphi \wedge \neg\beta\}}$$

Consequence Rule:
$$\frac{\varphi \Rightarrow \varphi'\ ,\ \{\varphi'\}\ P\ \{\psi'\}\ ,\ \psi' \Rightarrow \psi}{\{\varphi\}\ P\ \{\psi\}}$$

Assignment Rule:
$$\frac{\varphi \Rightarrow \psi(t/x)}{\{\varphi\}\ x \leftarrow t\ \{\psi\}}$$

As an example we will prove that the assignment rule

$$\frac{\varphi \Rightarrow \psi(t/x)}{\{\varphi\}\ x \leftarrow t\ \{\psi\}}$$

is valid.

Suppose the precondition $\varphi \Rightarrow \psi(t/x)$ is true, i.e. for all states s : if φ holds in s, so does $\psi(t/x)$.

We need to show $\{\varphi\}\ x \leftarrow t\ \{\psi\}$.

Consider a state s in which φ is true. Then, by the precondition, we know that $\psi(t/x)$ holds.

After executing the assignment, x has value t.

Thus, the result when evaluating predicate formula ψ in the follow-up state s' is the same as the value of $\psi(t/x)$ in state s, which was true. □

Example: suppose Q is a unary predicate, then

$$\{Q(i)\}\ i \leftarrow i-1\ \{Q(i+1)\}$$

Here, $\varphi = Q(i)$, $\psi(i) = Q(i+1)$, and $\psi((i-1)/i) = Q(i-1+1) = Q(i)$

Proving While Loops Correct

The While Rule suggests how we can verify (and develop) while loops. The general situation looks like this:

```
{ φ0 }          // precondition
P0              // initialization
{ φ }           // loop invariant
[ t ]           // runtime bound
while β do
   { φ ∧ β }
   P            // P terminates if φ ∧ β holds
   { φ }        // loop invariant holds again
end
{ φ ∧ ¬β }      // loop invariant + loop exit condition
{ ψ }           // postcondition
```

By finding a loop invariant that together with the loop exit condition implies the postcondition, we can establish the partial correctness of while loops.

But how do we know the while loop eventually stops?

In general this question can't be answered, but in many cases we can define a progress measure that allows us to prove termination.

Termination Lemma:

1. If P be a program that stops if $\varphi \wedge \beta$ holds, and

2. there is a function $t : S \to \mathbb{N}$ that maps program states to natural numbers with

$$\{\varphi \wedge \beta \wedge (t(s) = y)\}\ P\ \{\varphi \wedge (t(s) < y)\}$$

where s is the current state and y is a new variable that doesn't occur elsewhere, then

$$\texttt{while}\ \beta\ \texttt{do}\ P\ \texttt{end}$$

terminates when started in a state s in which φ holds.

Proof: Suppose φ holds in state s. If $\neg\beta$ also holds, then the claim is true.

Now assume β holds and suppose above while-loop does not terminate.

This means that for all $n \geq 0$ after the n-th iteration a state s_n is reached ($s_0 = s$) in which $\varphi \wedge \beta$ is true (proof by induction using condition 1.)

Therefore (condition 2.), $t(s_0) > t(s_1) > t(s_2) > \ldots$ and $t(s_n) \geq 0$ for all $n \geq 0$ — a contradiction. □

The following list summarizes the steps for verifying the correctness of while loops based on the While Rule and the Termination Lemma:

While Loop Verification Checklist

First, identify a suitable loop invariant φ and a runtime bound $t : S \to \mathbb{N}$. Then:

1. prove that φ holds when reaching the while statement for the first time,
2. prove $\{\varphi \wedge \beta\}\ P\ \{\varphi\}$,
3. prove $(\varphi \wedge \neg\beta) \Rightarrow \psi$,
4. prove that if $\varphi \wedge \beta$, the t-value > 0,
5. prove if $\varphi \wedge \beta$ holds, then executing P decreases the t-value

Example

We are looking for a program P that given $x \in \mathbb{N}$ computes $\lfloor\sqrt{x}\rfloor$.

E.g. $\lfloor\sqrt{0}\rfloor = 0$, $\lfloor\sqrt{1}\rfloor = 1$, $\lfloor\sqrt{2}\rfloor = 1$, $\lfloor\sqrt{3}\rfloor = 1$, $\lfloor\sqrt{4}\rfloor = 2$

$\{\ x \geq 0\ \}$ // precondition
P
$\{\ w = \lfloor\sqrt{x}\rfloor\ \}$ // postcondition

Analysis:

Previously we proved for $w \in \mathbb{N}$:

$$w = \lfloor\sqrt{x}\rfloor \ \Leftrightarrow\ w \leq \sqrt{x} < w + 1$$

which is equivalent to

$$w \geq 0\ \wedge\ w^2 \leq x < (w+1)^2$$

This suggests a loop that increments w until this condition is met.

We can use the first and second inequalities as loop invariant and the third inequality as loop exit condition.

$\{\ x \geq 0\ \}$ // precondition
$w \leftarrow 0$
$\{\ w \geq 0\ \wedge\ w^2 \leq x\ \}$ // loop invariant
$[\ t : x - w\ ]$ // runtime bound
`while` $(w+1)^2 \leq x$ `do`
  $\{\ w \geq 0\ \wedge\ w^2 \leq x\ \wedge\ (w+1)^2 \leq x\ \}$
  $w \leftarrow w + 1$
  $\{\ w \geq 0\ \wedge\ w^2 \leq x\ \}$ // loop invariant holds again
`end`
$\{\ w \geq 0\ \wedge\ w^2 \leq x < (w+1)^2\ \}$ // loop inv. + loop exit cond.
$\{\ w = \lfloor\sqrt{x}\rfloor\ \}$ // postcondition

Correctness Proof:

1.+2.+3: analysis on previous page and assignment rule applied twice.

4.: Prove: $(\varphi \wedge \beta) \Rightarrow t > 0$

$w \geq 0\ \wedge\ w^2 \leq x\ \wedge\ (w+1)^2 \leq x$

Because $w + 1 \geq 1,\ x \geq w + 1\ \Rightarrow x > w$
$\Rightarrow\ t = x - w > 0$

5.: prove that executing P decreases t: trivial, as w is increased.

Final Exam:

Friday, Dec. 17, 2pm in the Universiade Pavilion

Read sign for row, seats

2 hours, closed book

Format similar to term exams

Everything is relevant: all lecture notes, assignments, seminars

Study assignment and seminar solutions

There will be problem-solving questions

Monday, Dec. 13 Office Hour @ 3pm

Good luck!

FIN