Lecture 7: Program correctness

CAB203 Discrete Structures

Matthew McKague

Queensland University of Technology

cab203@qut.edu.au

Outline

Readings

This week, no readings.
Next week:

- Pace: 5.1 to 5.3, 6.1.4, 6.2

Outline

Induction basics

For $\mathbb{Z}^+$ we have a special property. For any predicate $P(x)$

$$\left[P(1) \wedge (\forall x \in \mathbb{Z}^+ \ (P(x) \rightarrow P(x+1)))\right] \rightarrow \forall x \in \mathbb{Z}^+ \ P(x)$$

What does this mean? Suppose we want to prove $P(x)$ for all $x \in \mathbb{Z}^+$

- *Base case:* Show that $P(1)$ is true (the first domino falls)
- *Inductive step:* Show that if $P(x)$ is true, then $P(x+1)$ is true (if this domino falls, then the next domino will fall)
- Conclude that $P(x)$ must be true for all $x \in \mathbb{Z}^+$ (all dominos must fall)

Induction example

Let us prove that the sum of the first n even numbers is $n(n+1)$.

Base case ($n = 1$): The first even number is 2. No sum here, just one even number.

$$2 = 1(1+1)$$

Induction example

Inductive step: For this step we assume that the first n even numbers sum to $n(n+1)$. That is,we assume

$$\sum_{j=1}^{n} 2j = n(n-1)$$

We want to show that the sum of the first $n+1$ even numbers is $(n+1)(n+2)$. The sum of the first $n+1$ even numbers is:

$$\begin{aligned}
\sum_{j=1}^{n+1} 2j &= 2(n+1) + \sum_{j=1}^{n} 2j \\
&= 2n + 2 + n(n+1) \\
&= 2n + 2 + n^2 + n \\
&= n^2 + 3n + 2 \\
&= (n+1)(n+2)
\end{aligned}$$

Outline

Outline

Bugs!

Programming is often more about fixing bugs than creating new code!

- Can we use logic to *prove* that code is correct?
- What do we even mean by correct?

Parts of a program

In a procedural language we have two main parts:

- the program's state: the contents of all variables, arrays and other data types that the program uses
- the program's code: a sequence of statements that can change the program's state

What do we mean by "correct"?

Before we can prove a program is correct, we need to specify what it should do:

- identify some variables as the inputs of the program (others initialised to constants)
- state some conditions that the inputs will satisfy (pre-conditions)
- identify some variables as the outputs of the program
- state some conditions that the outputs should satisfy (post-conditions)

We say that the program is correct if we can show that the post-conditions will always be true.

Blocks and assertions

Programs are complex, so we break them down into *blocks*.

- A *block* is a sequence of one or more code statements.
- *pre-conditions* are predicates about the state of the variables before a block is run
- *post-conditions* are predicates about the state after a block is run
- *assertions* refers to both pre- and post-conditions

This paradigm, and the rules we will see for using it, are called Floyd-Hoare logic.

Assertion basics

We write down pre- and post-conditions and blocks like so:

$\{x \geq 0\}$	pre-condition
`x = x + 1`	block of code 1
$\{x \geq 1\}$	post-condition for block 1 and pre-condition for block 2
`y = x / 2`	block of code 2
$\{y \geq 0.5\}$	post-condition for block 2

Here we have more than one block chained together with assertions interspersed.

Formal meaning of pre-/post-conditions and blocks

Assertions together with blocks form IF. . . THEN statements.

$\{P\}$
B
$\{Q\}$

Means "If P is true, and B is run, then Q will afterwards be true."

The usual rules for $\rightarrow$ apply. If P is false then the statement is trivially true. If we have multiple pre- or post-conditions then formally they are all ANDed together into one proposition.

Bad example

Because the state can change, the truth of an assertion can change!

$\{x = 0\}$	Assume this is true
$\{x \geq 0\}$	This will also be true
`x = x + 1`	state changes
$\{x = 0\}$	This is now false! Not a valid post-condition
$\{x \geq 0\}$	but this is still true

We need rules about how to find true post-conditions.

Post-conditions rule 1

If $p(x)$ is true before a block, and the block does not change x, then $p(x)$ can be taken as a post-condition.

- Typically this means that x is not on the left hand side of an assignment (but this depends on the language).
- We can generalise this to as many variables as we like ($p(x, y, \ldots)$) *so long as none of them change*.

Formally, this is special case of the assignment rule, covered later.

Implication rule

We can use logical equivalence and logical implication to add new assertions. We can also add any true propositions from mathematics.
Example:

$\{x \geq 0\}$	pre-condition
$\{\forall x\ (x \geq 0 \rightarrow x + 1 \geq 0)\}$	true predicate from mathematics
$\{x + 1 \geq 0\}$	logical implication

Basically, we can use logic and math, as long as we don't cross any blocks of code.

Assignment rule

One way the program state can change is by assignment:

`y = E`

If $p(E)$ is a pre-condition, then we can write $p(y)$ as a post-condition.

$\{E \geq 0\}$	pre-condition
`y = E`	assignment
$\{y \geq 0\}$	post-condition from assignment rule

Note that E could be an expression like `a * 3 + 5`.

A baby example

$\{x \geq 0\}$	pre-condition
$\{\forall x\,(x \geq 0 \rightarrow x + 1 \geq 0)\}$	true statement from mathematics
$\{x + 1 \geq 0\}$	logical implication
`y = x + 1`	some program code
$\{y \geq 0\}$	assignment rule
$\{x \geq 0\}$	x not affected by statement

Anatomy of a conditional statement

Conditional statements like `if` create different situations with different states.

`if A(x):`	$A(x)$ is the *condition*
some statements	this is the *if block*
`else:`	
some statements	this is the *else block*

Choice rule

If there is an `if` statement, branching on condition $A(x)$:

- Any pre-conditions of the `if` statement are pre-conditions of the if and else blocks
- In the if block we can add pre-condition $A(x)$
- In the else block we can add pre-condition $\neg A(x)$
- If $q(x)$ is a post-condition of both the if and else blocks we can add $q(x)$ as a post-condition to the whole thing

We must assume that evaluating $A(x)$ does not change the program's state (frequently not true in many languages!)

Choice rule

$\{p(x)\}$	pre-condition
`if A(x):`	
$\{p(x)\}$	bring the pre-condition forward
$\{A(x)\}$	$A(x)$ must be true if we are here
code ...	
$\{q(x)\}$	derived post-conditions
`else:`	
$\{p(x)\}$	bring pre-condition forward
$\{\neg A(x)\}$	$\neg A(x)$ must be true if we are here
code ...	
$\{q(x)\}$	derived post-conditions
$\{q(x)\}$	$q(x)$ was true in both branches

Choice rule example

`if` $x \geq 0$`:`	
$\{x \geq 0\}$	branch condition
`y = x`	
$\{y \geq 0\}$	assignment rule
`else:`	
$\{x < 0\}$	negate branch condition
$\{\forall x\,(x < 0 \rightarrow -x \geq 0)\}$	true from mathematics
$\{-x \geq 0\}$	logical implication
`y = -x`	
$\{y \geq 0\}$	assignment rule
$\{y \geq 0\}$	choice rule

Loops

Let's look at a `while` loop:

`while A(x):`	loop with condition $A(x)$
some statements	body of loop
rest of the program	

We need a special rule to deal with this.

Invariants

Suppose that:

$\{Q\}$
$\{P\}$
B block of code
$\{P\}$

is a valid proof of correctness for block B. The we can call P an *invariant* of B given Q as a pre-condition.

Example invariant

$\{x \geq 0\}$	pre-condition
$\{y \geq 0\}$	pre-condition
$\{xy \geq 0\}$	from math and logic
`y = y * x`	
$\{y \geq 0\}$	assignment rule

Here $y \geq 0$ is an invariant. Notice that $y \geq 0$ is not an invariant if we don't have $x \geq 0$ as a pre-condition.

Loop rule

Suppose that P is an invariant of B given $A(x)$. Then the following is a valid proof of correctness.

$\{P\}$	invariant is true before `while`
`while A(x):`	$A(x)$ is the condition
B	block of code
$\{P\}$	invariant still true after loop
$\{\neg A(X)\}$	condition now false to exit the loop

Loop example

$\{y \geq 0\}$	pre-condition
$\{x \geq 0\}$	invariant as a pre-condition
`while y >= 0:`	
$\{y \geq 0\}$	loop condition holds
$\{x \geq 0\}$	invariant
$\{xy \geq 0\}$	logic and math
`x = x * y`	
`y = y - 1`	
$\{x \geq 0\}$	assignment rule
$\{x \geq 0\}$	invariant still true after loop
$\{y < 0\}$	condition now false to exit the loop

Here I'm showing that $x \geq 0$ is an invariant inside the body of the loop.

Outline

Loop termination

Invariants allow us to conclude correctness in loops, but they don't say anything about whether we ever exit the loop. Example:

$\{y \geq 0\}$	pre-condition
$\{x \geq 0\}$	invariant as a pre-condition
`while y >= 0:`	
$\{y \geq 0\}$	loop condition holds
$\{x \geq 0\}$	invariant
$\{xy \geq 0\}$	logic and math
`x = x * y`	
`y = y + 1`	$\leftarrow$ y never goes down
$\{x \geq 0\}$	assignment rule
$\{x \geq 0\}$	invariant still true after loop
$\{y < 0\}$	condition now false to exit the loop

Here everything is fine, but the loop will never terminate.

Total correctness

- The loop rule shows correctness *assuming the loop terminates*, this is called *partial correctness*
- The loop rule does not show that the loop exits
- We need an additional structure to show that the loop terminates
- *Total correctness* means partial correctness and that the loop terminates

Loop termination

How to show a loop terminates? One way:

- Define some variable x that starts at some positive natural number
- Show that x decreases by at least 1 each time through the loop
- Show that x must always be at least 1

If the loop never terminated, then this would produce a contradiction: x must keep decreasing, and yet must always stay above 1!

In general, it is a very hard problem to decide if a given program terminates. This is called the halting problem and it is undecidable.

Loop variants

Suppose that:

$\{Q\}$
$\{P\}$
$\{0 \leq V = z\}$
B block of code
$\{P\}$
$\{0 \leq V \leq z - 1\}$

is a valid proof of correctness for block B for all $z \in \mathbb{Z}$. Then V is called a *loop variant* of B. Note that P is an invariant of B given Q.

Imagine running B repeatedly. P and $0 \leq V$ must always be true since B can't change them. But eventually $z - 1 < 0$. The *only* thing that can change is Q can become false.

Loop variants

Suppose that P is an invarint and V is loop variant of B, assuming Q. Then the following is a valid proof of correctness

$\{P\}$
$\{0 \leq V\}$
while Q:
    B block of code
$\{P\}$
$\{\neg Q\}$

We know from the loop variant that Q must eventually be false, so the loop will terminate. Correctness comes from the invariant P.

Example: summing an array

We will show that the following program to sum up the contents of the array A[ ] is correct:

```
k = 0
s = 0
while k ≤ n − 1:
        t = s + A[k + 1]
        s = t
        k = k + 1
```

Loop variants and invariants

$//k$ will start at 0	
$\{k \leq n-1\}$	This will be the loop condition
$\{s = \sum_{j=1}^{k} A[j]\}$	loop invariant
$\{0 \leq n-k = z\}$	$n-k$ is the loop variant
	$0 \leq n-k$ is also an invariant
$\mathrm{t} = \mathrm{s} + \mathrm{A[k+1]}$	
$\{t = \sum_{j=1}^{k+1} A[j]\}$	assignment rule plus math
$\mathrm{s} = \mathrm{t}$	
$\{s = \sum_{j=1}^{k+1} A[j]\}$	assignment rule
$\mathrm{k} = \mathrm{k}+1$	
$\{s = \sum_{j=1}^{k} A[j]\}$	assignment rule
$\{n-k = z-1\}$	assignment rule plus math
$\{k \leq n\}$	assignment rule plus math
$\{0 \leq n-k\}$	math
$\{s = \sum_{j=1}^{k} A[j]\}$	loop invariant
$\{0 \leq n-k = z-1\}$	$0 \leq n-k$ is also an invariant

Example: summing an array

$\{0 = \sum_{j=1}^{0} A[j]\}$	math
k = 0	
s = 0	
$\{s = \sum_{j=1}^{k} A[j]\}$	assignment rule
$\{0 \leq n - k\}$	assignment rule and math
while $k \leq n - 1$:	
t = s + A[k + 1]	
s = t	
k = k + 1	
$\{s = \sum_{j=1}^{k} A[j]\}$	loop invariant
$\{0 \leq n - k\}$	loop invariant
$\{k > n - 1\}$	negation of loop condition
$\{k = n\}$	math
$\{s = \sum_{j=1}^{n} A[j]\}$	math

Outline

Useful assertions

We need to define pre- and post-conditions that outline the task that our program should accomplish.
Example, factoring:

inputs: $\mathtt{x}$	variable to store the input
$\{x \in \mathbb{N}\}$	basic properties of x
$\{\exists a, b \in \mathbb{N}\ (x = ab \wedge a \neq n \wedge b \neq n)\}$	x is composite
...	
our code	
assign to y and z	
...	
outputs: $\mathtt{y}$ and $\mathtt{z}$	decide what the outputs are
$\{x = yz \wedge x \neq y \wedge x \neq z \wedge y, z \in \mathbb{N}\}$	code is correct if the outputs are factors of x

Sort example

We can verify that largish parts of programs work correctly.
Example, a sort function:

input: `A[], n` — input is an array of integers
$\{n \in \mathbb{N}\}$
$\{\forall j \in \{1 \ldots n\}\ (A[j] \in \mathbb{Z})\}$
...
code here
...
output: `A[]`
$\{\forall j \in \{1 \ldots n-1\}\ (A[j] \leq A[j+1])\}$ — array is sorted

Later we can use this proof wherever the sort function is used.
Similarly, we can prove that implementations of data structures are correct.

Declarative and imperative syntax

- Procedural languages are *imperative*: they tell the computer to *do* something.
- Assertions are *declarative*: they don't do anything, but describe properties of data.

A proof of program correctness incorporates both imperative elements (program statements) and declarative elements (assertions)

Functional languages

There exist programming languages which are entirely declarative:

- CSS
- Haskell
- Lisp
- ML
- ...

These languages don't tell the computer *how*, they only describe conditions on what should happen. The compiler determines the how.

Functional languages and program correctness

If a functional languages is already declarative, do we need program correctness?

- Functional language philosophy is to build programs out of small, easy to understand functions that are "obviously" correct.
- But functional programs can still be wrong if the description is wrong
- Much of the value of a proof of correctness is in forcing a careful second consideration of the code
- There exist languages that are designed to *check* proofs. Coq is an example

Uses of program correctness

Proving a program correct is long and hard! So it is usually used only for

- Safety critical code
- Security critical code
- Example: NICTA in Sydney proved the correctness of an operating system kernel written in 7500 lines of C code.
- For larger programs, testing-driven development is becoming more popular: it offers a different sort of second perspective on a programming problem.
- Most programs are still just written and bugs fixed when they are found