IP Lecture 3: More on Invariants

Joe Pitt-Francis

—with thanks to Mike Spivey & Gavin Lowe—

A utility function for adding up milk

```
/** Calculate sum of a
  * Post: returns sum(a) */
def findSum(a : Array[Int]) : Int = {
  val n = a.size
  var total = 0; var i = 0
  // Invariant I: total = sum(a[0..i)) && 0<=i<=n
  while(i < n){
    // I && i<n
    total += a(i)
    // total = sum(a[0..i+1)) && i<n
    i += 1
    // I
  }
  // I && i=n
  // total = sum(a[0..n))
  total
}
```

The main program

We want to be able to provide the arguments on the command line, e.g.

```
> scala Milk 3 1 4 0 3
```

When extra arguments are provided on the command line, they are copied into the parameter `args` of the main function

```
def main(args : Array[String])
```

Note that these are `String`s; we will need to convert them into `Int`s in order to use `findSum`. If `s` is a `String` then `s.toInt` gives the `Int` that it represents (if it does represent an `Int`).

The main function

Here's some code that (1) calculates the size of `args`; (2) initialises a new array `a` of `Int`s of the same size; (3) converts each element of `args` into an `Int` and copies it into `a`; (4) calls `findSum` on `a`.

```
def main(args: Array[String]) = {
  val n = args.size
  val a = new Array[Int](n)
  for(i <- 0 until n) a(i) = args(i).toInt
  println(findSum(a))
}
```

`for` loops

The for-loop `for(x <- xs) body` executes `body`, where `x` takes each value in the sequence `xs` in turn.

`0 until n` creates the sequence of numbers from `0` to `n-1`, inclusive, which is sometimes written as `[0..n)`.

Similarly, `0 to n` creates the sequence of numbers from `0` to `n`, inclusive.

We could have written the main loop of the `findSum` function as

```
for(i <- 0 until n) total += a(i)
```

and the main loop of the `fact` function as

```
for(i <- 1 to n) f *= i
```

for loops versus while loops

The for loop

```
for(i <- a until b) Body
```

is equivalent to

```
var i = a
while(i<b){ Body; i = i+1 }
```

assuming Body doesn't change i.

Invariants for `for` loops

Going the other way, using invariant $I \wedge$ `a` $\leq$ `i` $\leq$ `b`:

```
Init            // I[a/i]
var i = a       // I
while(i<b){     // I && a<=i<b
  Body          // I[i+1/i]
  i = i+1       // I
}               // I[b/i]
```

(where `I[x/i]` means `I` with `x` substituted for `i`) is equivalent to

```
Init                      // I[a/i]
for(i <- a until b){      // I && a<=i<b
  Body                    // I[i+1/i]
}                         // I[b/i]
```

Invariants are easier to reason about using `while` loops than `for` loops, so we will tend to favour `while` loops from now on (except for trivial loops). Also, `while` loops are normally more efficient than `for` loops.

Using `map` and anonymous functions

Recall the code

```
val n = args.size
val a = new Array[Int](n)
for(i <- 0 until n) a(i) = args(i).toInt
```

This should remind you of `map` from functional programming; `Scala` has a `map` operation over arrays, so we could have written this as

```
val a = args.map(_.toInt)
```

Here, "`_.toInt`" is the function that applies `toInt` to its argument; the underscore represents a "hole" where the argument is inserted. We could also have written this as `x => x.toInt`.

When using either of these notations for anonymous functions, it is sometimes necessary to provide the type of the argument, e.g. `((_:String).toInt)` or `((x:String) => x.toInt)`.

Using `map`

If `xs` is an array, then `xs.map(f)` gives back a new array resulting from applying `f` to each element of `xs`.

Here we want to apply `map` with the function that, given string `s`, returns `s.toInt`. We can write this function anonymously as either `(s => s.toInt)` or as `(_.toInt)`.

Here's a slicker version of `main`:

```
def main(args : Array[String]) = {
  val a = args.map(_.toInt)
  println(findSum(a))
}
```

or even (perhaps less clearly):

```
def main(args : Array[String]) = println(findSum(args.map(_.toInt)))
```

Exponentiation

We will develop three programs to calculate `x^n` (i.e. x^n), where `x` and `n` are supplied by the user.

- We'll take `x` to be a `Double`, i.e. a 64-bit floating point number;
- We'll take `n` to be a non-negative `Long`, i.e. a 64-bit integer (so we can experiment with large values of `n`).

A recursive definition

Here's a straightforward recursive definition.

```
/** Calculate x^n
  * Pre: n >= 0
  * Post: returns x^n */
def exp(x: Double, n: Long) : Double =
  if(n==0) 1 else x*exp(x,n-1)
```

How much time does `exp` take?

How much space does `exp` take?

Getting the arguments

We will arrange for the user to supply the arguments on the command line, typing, for example

`scala RecExp 1.34 45`

We expect there to be precisely two arguments, i.e. `args.length==2`.

We can convert the arguments to the correct type as:

```
val x = args(0).toDouble
val n = args(1).toLong
```

We expect `n` to be non-negative.

The complete program

```
object RecExp{
  /** Calculate x^n
    * Pre: n >= 0
    * Post: returns x^n */
  def exp(x: Double, n: Long) : Double =
    if(n==0) 1 else x*exp(x,n-1)

  def main(args: Array[String]) = {
    if(args.length != 2) println("Usage: scala RecExp x n")
    else{
      val x = args(0).toDouble
      val n = args(1).toLong
      if(n>=0) println(x+"^"+n+" = "+exp(x,n))
      else println("Error: second argument should be non-negative")
    }
  }
}
```

An iterative definition

```
/** Calculate x^n
  * Pre: n >= 0
  * Post: returns x^n */
def exp(x: Double, n: Long) : Double = {
  require(n>=0)
  // Invariant I: y = x^i && i <= n
  var y = 1.0; var i = 0 // I established
  while(i<n){
    y = y*x; i = i+1
  }
  // I && i=n, so y=x^n
  y
}
```

Correctness

Remember the rules for proving correctness using an invariant `I`.

```
// pre
Init
// I
while(test){
  // I && test
  Body
  // I
}
// I && not test
// post
```

We need to check:

- `Init` establishes `I`, assuming `pre`;
- `Body` maintains `I`;
- `I && not test` implies `post`;

And to identify a variant `v` such that

- `v` is integer valued, assuming the invariant;
- `v` is at least `0`, assuming the invariant;
- `v` is decreased at each iteration.

Why do these hold here?

Patterns of invariants

Here are the postconditions and invariants we've seen so far

Program	Postcondition	Invariant
`fact`	`f = n!`	`f = i! && i <= n`
`findSum`	`total =` $\sum$`a[0..n)`	`total =` $\sum$`a[0..i) && 0 <= i <= n`
`exp`	`y = x^n`	`y = x^i && i <= n`

Spot the pattern?

Complexity

How much time does `exp` take?

How much space does `exp` take?

Towards a faster version

We will develop a faster version based on the invariant

$$I \mathrel{\widehat{=}} \mathtt{y} * \mathtt{z}^{\mathtt{k}} = \mathtt{x}^{\mathtt{n}} \wedge 0 \leq \mathtt{k} \leq \mathtt{n}$$

with variant `k`.

We can establish the invariant using

```
var y = 1.0; var z = x; var k = n
```

We want a loop of the form

```
while(k>0){
  // I && k>0
  ...
  // I
}
// I && k=0, so y=x^n
y
```

The main loop

Our code will test if `k` is even, using the test `if(k%2 == 0)`.

If `k` is even, we can calculate as follows:

```
y * z^k  =  y * z^(2 * k/2)
         =  y * (z^2) ^ (k/2)
         =  y * (z*z) ^ (k/2)
```

which shows that the code

```
z = z*z; k = k/2
```

maintains the invariant.

The main loop

If `k` is odd, we can calculate as follows:

```
y * z^k  =  y * z^(1 + 2 * k div 2)
         =  y * (z * (z^2) ^ (k div 2))
         =  (y * z) * (z*z) ^ (k div 2)
```

which shows that the code

```
y = y*z; z = z*z; k = (k-1)/2
```

maintains the invariant.

(The “/” here is integer division, i.e. `div`, so we could have set `k = k/2`.)

Fast exponentiation

```
/** Calculate x^n.
  * Pre: n >= 0
  * Post: returns x^n */
def exp(x:Double, n:Long) : Double = {
  require(n>=0)
  // Invariant I: y * z^k = x^n && 0 <= k <= n.  Variant: k
  var y = 1.0; var z = x; var k = n
  while(k>0){
    if(k%2==0){
      z = z*z; k = k/2
    }
    else{
      y = y*z; z = z*z; k = k/2
    }
  }
  // I and k==0 =>  y = x^n
  y
}
```

Complexity

How much time does `exp` take?

How much space does `exp` take?

Summary

- Command line arguments;
- `for` loops;
- `map` and anonymous functions;
- A pattern for invariants: replace a constant with a variable;
- Comparing three different programs for exponentiation.
- Next time: Testing.