IP Lecture 2: Invariants

Joe Pitt-Francis

—with thanks to Mike Spivey & Gavin Lowe—

Reminder: Compile and run the Factorial object

```
> scalac Factorial.scala
# or better...
> fsc Factorial.scala

> scala Factorial
```

- Interpreted languages (BASIC and MATLAB) take and evaluate one line of human-readable code at a time. (User-friendly?)
- Compiled languages (C++ and FORTRAN) convert the human-readable program text into an executable binary file to be run on a *specific* machine. (Run faster?)
- Intermediate languages (Java and Scala) compile into *machine-neutral* bytecode (`.class` files) which are then executed by the Java Virtual Machine using a just-in-time compiler. (Best of both worlds?)

A reminder of the Factorial code with recursion

```
object Factorial{
  /** Calculate factorial of n
    * Pre: n >= 0
    * Post: returns n! */
  def fact(n: Int) : BigInt = {
    require(n>=0)
    if(n==0) 1 else fact(n-1)*n
  }

  // Main method
  def main(args: Array[String]) = {
    print("Please input a number: ")
    val n = scala.io.StdIn.readInt
    // if (n>=0){...
    val f = fact(n)
    println("The factorial of "+n+" is "+f)
  }
}
```

Calculating factorials using a `while` loop

```
/** Calculate factorial of n
  * Pre: n >= 0
  * Post: returns n! */
def fact(n: Int) : BigInt = {
  require(n>=0)                         // assume n>=0
  var f: BigInt = 1; var i = 0          // f = i! and i<=n
  while(i<n){                           // f = i! and i<n
    i = i+1                             // f = (i-1)! and i<=n
    f = f*i                             // f = i! and i<=n
  }
  // f = i! and i = n, so f = n!
  f
}
```

(The rest of the program is unchanged.)

Syntactic notes

- Two commands can be written on one line by separating them with a semicolon ("`;`"), as on line 6.
- The code `while(test) body` does the following
  1. evaluates `test`;
  2. if `test = true`, executes `body`, and returns to step 1;
  3. if `test = false`, finishes.
- Here the body of the `while` loop contains more than one command, so has to be enclosed in curly brackets (if the body were a single command, the brackets would be unnecessary).

Invariants

The previous code illustrated an important pattern for proving properties of `while` loops, namely the idea of an invariant.

An invariant is a property that is true at the start and end of each iteration of the loop.

In the previous code, the property `f = i! and i <= n` was an invariant. (As noted on lines 6, 9, etc.)

Aside: calculating factorials using a `for` loop

Here's another implementation of `fact`. It also uses `f`, that starts at `1`, and is multiplied by each number `i` from `1` to `n` (inclusive).

```
1  // Calculate factorial of n, using a for loop
2  // Precondition: n >= 0
3  def fact(n:Int) : BigInt = {
4    require(n>=0)
5    var f : BigInt = 1  // f = 0!
6    for(i <- 1 to n){   // f = (i-1)!
7      f = f*i           // f = i!
8    }
9    // f = n!
10   f
11 }
```

We cannot say `// f = i!` on line 5. 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).

Invariants

The normal pattern for a program with an invariant `I` is as follows:

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

We need to check:

- `Init` establishes `I`, assuming `pre`;
- `Body` maintains `I`;
- the loop terminates (see later);
- `I and not test` implies `post`.

The milk bill example

Imagine we're trying to add up a milk bill. We've got a sequence containing the number of bottles of milk delivered each day, and we want to calculate the total.

Of course, this is an instance of a general problem: adding up a sequence of numbers.

We will develop correct code, using an invariant to guide us.

Arrays

Scala has lots of sequence-like (or list-like) datatypes. Perhaps the most common type is arrays.

If `T` is a type then `Array[T]` represents the type of arrays that hold data of type `T`. (In Scala, square brackets are always used for parametric polymorphism.)

Each array has a fixed size. The `i`th entry of array `a` can be obtained using `a(i)`. (Note the round parentheses!) The indexing is zero-based, i.e., if `a` is of size `n`, then the elements are `a(0),...,a(n-1)`. This indexing operation takes constant time (unlike the corresponding operation on Haskell lists).

Arrays are mutable. The individual entries can be updated, e.g.

```
a(i) = a(i) + 1
```

The postcondition

We want to develop a function

```
def findSum(a : Array[Int]) : Int = ...
```

which returns the sum of the entries in `a`. If `a` has `n` entries, then the requirement is to calculate `total` such that

$$\texttt{total} = \sum \texttt{a}[0..\texttt{n})$$

(In comments in the actual code, we'll write "`sum`" for "$\sum$".)

We have postcondition:

$$\textbf{post: returns } \texttt{total} \textbf{ s.t. } \texttt{total} = \sum \texttt{a}[0..\texttt{n})$$

or more simply

$$\textbf{post: returns } \sum \texttt{a}[0..\texttt{n})$$

The precondition is simply *true*, so we omit it.

The invariant

Our program will add up the elements of `a` in order. Let's use a variable `i` to record how far we've got. Then at each point we will have

$$\mathtt{total} = \sum \mathtt{a}[0..\mathtt{i}) \wedge 0 \leq \mathtt{i} \leq \mathtt{n}$$

Let's call this invariant "I".

The size of `a`

We can set `n` to be the size of `a` by

```
val n = a.size
```

Aside: `Arrays` are objects. When an object `obj` provides an operation `op`, then it is invoked by

```
obj.op
```

If the operation takes argument `x`, then it is invoked by

```
obj.op(x)
```

Initialization

Recall the invariant

$$\texttt{total} = \sum \texttt{a}[0..\texttt{i}) \land 0 \le \texttt{i} \le \texttt{n}$$

Our initialization needs to establish `I`. The following code does this.

```
val n = a.size
var total = 0; var i = 0
```

The main loop

When `i` gets to `n`, we'll have finished. So we are after a loop of the following form.

```
while(i < n){
  // I && i<n
  ...
  // I
}
```

which we can do as follows

```
    // I && i<n
    total = total + a(i)
    // total = ∑a[0..i+1) && i<n
    i = i + 1
    // I
```

Aside: some useful notation

It's quite common to have assignments of the form

```
v = v + exp
```

for variable `v` and expression `exp`. This can be abbreviated to

```
v += exp
```

Similar shorthands hold for other binary operators

The body of the loop can be abbreviated to

```
    total += a(i)
    i += 1
```

The complete function

```
/** 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
}
```

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`;
- the loop terminates.

We've already checked all of these except for termination.

Termination

For termination, the normal approach is to identify an expression `v` of the program variables, known as a variant, and to check

- `v` is integer valued, assuming the invariant;
- `v` is at least `0`, assuming the invariant;
- `v` is decreased at each iteration (maybe by more than one).

If these are true, the loop can do only a finite number of iterations before terminating.

Normally, the variant is a measure of how much more work needs to be done.

Here we take the variant to be `n - i`.

Hoare triples

The (mathematical) notation

$$\{P\}\texttt{Prog}\{Q\}$$

is used to mean that if program `Prog` is executed from a state that satisfies P, it is guaranteed to terminated in a state that satisfies Q.

P is called the precondition, and Q is called the postcondition.

Correctness, again

We can describe our correctness theorem as follows.

If

- $\{pre\}Init\{I\}$;
- $\{I \wedge test\}Body\{I\}$;
- $I \wedge \neg test \Rightarrow post$;
- $I \Rightarrow v \in \mathbb{N}$;
- $\{I \wedge test \wedge v = V\}Body\{v < V\}$ (where V is a logical constant);

then

$$\{pre\}\ Init\ ;\ while(test)Body\ \{post\}$$

Formal methods

- Invariants are useful for us to have more certainty in the correctness of our programs.
- Invariants and Hoare triples can be used to prove a program correct.
- They can also be used to systematically derive programs—by refining from a mathematical specification to a working program.
- Systematic refinements and proofs can be done by machine (Z and the B-method).

Iteration, invariants, recursion and induction

There is a sense in which recursion and iteration are related, because they perform similar tasks.

- Recursion: solve a large problem by breaking it up into smaller ones.
- Iteration: Start small and keep repeating until task is done.
- Prove recursion is correct with proof by induction.
- Prove iteration is correct with invariants.
- Need to show that finite recursion terminates and that loops terminate.

Summary

- Invariants;
- Using invariants for deriving programs;
- Proving termination;
- Hoare triples; preconditions and postconditions;
- Arrays;
- Next time: Making milk function into a program. More examples of invariants.