IP Lecture 4: Testing

Joe Pitt-Francis

—with thanks to Gavin Lowe—

Checking preconditions

The first version of `exp` (exponential function) had precondition `n` ≥ 0, but checked in the `main` function that the precondition was met:

```
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]) = {
     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")
  }
}
```

Later, we put the check into the `exp` function: `require(n>0)`.

If the precondition for a function is not met, then what should we do?

Valid inputs

The real answer to the question on the previous slide is —
"It depends".

It's good to have a hierarchy of errors

Level 1 If the error can be fixed safely, then fix it. If need be, warn the user.

Level 2 If the error could be caused by user input then throw exception up to calling code, since the calling code should have enough context to fix the problem.

Level 3 If the error should not happen under normal circumstances then trip an assertion.

(So far we have been using `assert` or `require` in a "stop the world" sense. We could have caused a negative exponent to trigger the calling of a different function "p-th root".)

Scala Exceptions

`require` throws an `IllegalArgumentException` which can be caught and dealt with. (Lines 9 and 10 below.)

`assert` throws an `AssertionError` is meant to be taken more seriously.

```
1  // Attempts to use exp function without checking precondition.
2  // Does something else when precondition fails
3  def main(args:Array[String]) = {
4    val x = args(0).toDouble; val n = args(1).toLong
5    var ans = 0.0
6    try {
7       ans = exp(x,n)
8    } catch {
9      case iae: IllegalArgumentException =>
10       {println("Using library function"); ans = Math.pow(x,n)}
11     case ae: AssertionError => {/*Stop!*/}
12   }
13   println(x+"^"+n+" = "+ans);
14 }
```

Unit testing

Using invariants to develop code can greatly reduce the number of bugs, and increase our confidence in the code. However, testing is still necessary. Unit testing is testing performed on an individual unit of a program, such as a function. Writing unit tests also allows easy re-testing if you subsequently change your code. Of course, no feasible amount of testing is guaranteed to find all bugs (except for trivial code).

Unit testing

Here's a `main` method that tests `exp`.

```
def main(args: Array[String]) = {
  assert(exp(2.0,3) == 8.0)
  assert(exp(2.0,8) == 256.0)
  assert(exp(2.0,0) == 1.0)
}
```

`assert` raises an exception if its argument is false. We've seen that it's rather like `require`, but the latter is conventionally used for preconditions.

Problem: If a test fails then `assert` by default will "stop the world" and no other tests will be run.

However, it's normally better to use a proper testing framework, to allow tests to be re-run. ScalaTest is a testing framework for Scala[a].

[a] available from `http://www.scalatest.org/`.

ScalaTest

A ScalaTest script is simply a Scala program such as the following.

```
import org.scalatest.FunSuite
import FastExp.exp

class Tests0 extends FunSuite{
  test("2^3 = 8"){ assert(exp(2.0,3) == 8.0) }
  test("2^8 = 256"){ assert(exp(2.0,8) == 256.0) }
  test("2^0 = 1"){ assert(exp(2.0,0) == 1.0) }
}
```

The function `test` takes a string which names the test (in round parentheses), and some code defining the test (in curly brackets).

Compiling ScalaTest

To use ScalaTest, you need to have the ScalaTest files on your `CLASSPATH` environment variable. For example, add a line like

```
export CLASSPATH=${HOME}/Scala/scalatest_2.12-3.0.5.jar:$CLASSPATH
```

to your `.bashrc` file, then type `source .bashrc` at the shell prompt.

Notes:

- The most recent versions of ScalaTest depend on Scalactic
- You can experiment first by adding the `CLASSPATH` explicitly on the command line with

  ```
  fsc -cp ./scalatest_2.12-3.0.5.jar:./scalactic_2.12-3.0.5.jar ...
  ```

- `2.12` is the Scala version (which has to match Scala on your machine) and `3.0.5` is the Scalatest version number.

You can then compile the test suite using `fsc` or `scalac` in the normal way.

Using ScalaTest

You can then run the tests by typing:

```
scala org.scalatest.run Tests0
```

or something like

```
scala -cp ./scalatestxxx.jar org.scalatest.run Tests0
```

If you find that incantation difficult to remember then add the following to your `.bash_aliases` file:

```
alias scalatest='scala org.scalatest.run'
```

then type `source .bash_aliases`, and subsequently you can run the tests by typing:

```
scalatest Tests0
```

Using ScalaTest

If an assertion fails, the error message you get back might not be very useful. An assertion such as

```
test("2^3 = 8"){ assert(exp(2.0,3) === 8.0) }
```

(with three "=" signs) gives a better error message.

A test can also check that a function throws an exception, for example.

```
test("negative exponent"){
  intercept[IllegalArgumentException]{ exp(2.0,-1) } }
```

For more details see Section 14.2 of Programming in Scala or http://www.scalatest.org/getting_started_with_fun_suite.

ScalaTest semantics

When Scalatest encounters a `test`, it does not immediately run it, but adds it to a collection. When the rest of the code is finished, it then runs all the tests. So, for example, the following will give an error.

```
  var x = 0
  test("x=0"){ assert(x===0) }
  x = 1
  test("x=1"){ assert(x===1) }
}
```

Hence setting up an individual test should be included in the test:

```
  var x = 0
  //...
  test("x=1"){
    x=1
    assert(x===1)
  }
```

Floating point calculations

So far we have been able to give a definitive pass to tests:

```
test("2^8 = 256"){ assert(exp(2.0,8) === 256.0) }
```

Most of the tests we might want to try fail. This test

```
test("0.1^10 = 1E-10"){ assert(exp(0.1,10) === 1e-10) }
```

gives

```
- 0.1^10 = 1E-10 *** FAILED ***
  1.0000000000000011E-10 did not equal 1.0E-10 (ExpTests1.scala:16)
```

Why is this?

Floating point tolerances with Scalatic

A floating point equality tests almost never a good idea. Calculations can be off by a relative error *(machine epsilon* $\approx 2.2 \times 10^{-16}$*)* due to

- representation error (what's 0.1 in binary?); and
- rounding error (accumulated during calculation).

```
import org.scalatest.FunSuite
import FastExp.exp
import org.scalactic.Tolerance._

class ExpTests1 extends FunSuite{
  //...
  // Floating point comparison should allow for rounding
  test("0.1^10 = 1E-10"){  assert(exp(0.1,10) === 1e-10) }
  test("0.1^10 ~= 1E-10"){ assert(exp(0.1,10) === 1e-10 +- 1e-25) }
}
```

Is this a good tolerance for `FastExp`? For `SlowExp`?

Black box unit testing

Black box testing treats a component such as function or (later) an object as a "black box".

- Tests influenced by knowledge of component specification (what it's meant to do) and interface (what the arguments are).
- Tests have no knowledge about internal organization of a component (the code/data is hidden from the test).

Advantages:

- Stricter focus on specification and the user's point of view.
- Tests can be truly independent of the developers (e.g., better to test `FastExp` with `Math.pow` than to write `FastExp` again).
- Tests can be written as soon as the specification is known—before development work starts (c.f. Test-driven development).

White box unit testing

White box testing or "transparent box testing" has knowledge of a component's implementation. A test for an object might also have privileged access to its private data.

- Tests influenced by knowledge of components internals.
- Tests may check that invariants are not violated (in objects).

Advantages:

- Testing is more thorough, with the possibility of covering many different paths through the code (c.f. path coverage).
- Exposure of data allows us to check preconditions, invariants etc.
- Testing can start while the component is still being developed (before there is a graphical user interface)[a].

[a] In many projects testing/implementation starts before there is a specification!

Equivalence class testing

One testing technique is to divide all possible inputs into equivalence classes in which the behaviour of the component ought to be the same. Test one input from each equivalence class: assuming that because two inputs from the same class should behave the same then they actually will.

For example, inputs to `CalculateArgument` of a complex number might be split into 5 classes (each of the quadrants and the origin).

- Weak equivalence class testing: test one representative from each equivalence class for each input (the other inputs to the function can take any value).
- Strong equivalence class testing: test one representative in the Cartesian product.

If we look at `CalculateArgument` as having 2 inputs (real and imaginary) there could be $3 \times 3 = 9$ classes for strong testing.

Boundary value testing

Equivalence classes naturally give us boundaries: e.g. what happens to `CalculateArgument` when the input is close to the origin?

Dogma says that most code fails on inputs which are near to the boundaries. Testing should be concentrated near to the boundary: if precondition of a function is `age`≥ 16 then test that it gives an error at `age`$= 15$ and it's successful at `age`$= 16$.

To note:

- Equivalence classes and boundary testing sometimes ignores internal boundaries where equivalence classes are further divided by the implementation.
- Boundary testing works well with integers but not so well with floating point (how near do you need to be?) and not so well with generic equivalence tests (what's the boundary between Firefox-Linux and InternetExplorer-Windows?).

Debugging

If a test fails, how do you find the bug that caused it? Unit tests help you focus because they indicate which components have test suites which are now failing. There are a number of less sophisticated techniques.

Try adding `println` statements into your program, to show the state of variables. For example, in the loop in the `exp` procedure you could add a line:

```
println("exp: y = "+y+"; z = "+z+"; k = "+k+"; x = "+x+"; n = "+n)
```

If this produces too much output, you could try piping it through a pager like `less`:

```
scala FastExp 2 3 | less
```

or saving the output in a file:

```
scala FastExp 2 3 > log.txt
```

Debugging

You can test whether particular properties hold using an assertion. This also helps document your code.

For example you can test whether your claimed invariant really does hold using an assertion; for example (where `slowExp` is the slow, but trusted, implementation of exponentiation):

```
assert(y*slowExp(z,k) == slowExp(x,n), "exp invariant failed; y = "+y+...)
```

Of course, this will slow down the program.

You can also test whether your variant is really being decreased, saving the previous value of the variant in a variable, say `variant`; for example:

```
val oldVariant = variant
variant = ...
assert(variant<oldVariant, ...)
```

Debugging

If you're not sure which procedure is causing problems, you can try logging each procedure call, and return from a procedure; for example

```
println("Starting exp: x = "+x+"; n = "+n)
...
println("Finishing exp: result = "+y)
```

Also, explicitly testing the precondition and/or postcondition can help detect bugs.

But it's far easier to debug a procedure in isolation than after you've inserted the procedure into a larger program — hence the need for unit testing.

Some testing tips

1. Use a testing framework library, such as `ScalaTest`.
2. Add one or more tests for every new piece of functionality, no matter how small.
3. Make tests definitive—they should either pass or fail, but beware of floating point tolerances.
4. Write tests (and preconditions) for *corner cases*—collinear triangles, singular matrices, $0 + 0i$ etc.
5. Concentrate tests near the *boundary* between types of input.
6. Review your tests from time to time. Add new tests as necessary and remove only those which you know to be redundant.
7. Automate your testing, so that you do not have to remember.

Summarised from Pitt-Francis & Whiteley, *Guide to Scientific Programming in C++, 2nd edition* (2018).

Summary

- Exceptions and assertions;
- Unit testing with Scalatest (c.f. JUnit, CxxTest, unittest...);
- Black box and white box testing
- Debugging.
- Next time: string searching.