ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Functional Random Generators

Principles of Reactive Programming

Martin Odersky

Other Uses of For-Expressions

Operations of sets, or databases, or options.

Question: Are for-expressions tied to collections?

Answer: No! All that is required is some interpretation of `map`, `flatMap` and `withFilter`.

There are many domains outside collections that afford such an interpretation.

Example: random value generators.

Random Values

You know about random numbers:

```
import java.util.Random
val rand = new Random
rand.nextInt()
```

Question: What is a systematic way to get random values for other domains, such as

- booleans, strings, pairs and tuples, lists, sets, trees

?

Generators

Let's define a trait Generator[T] that generates random values of type T:

```
trait Generator[+T] {
  def generate: T
}
```

Some instances:

```
val integers = new Generator[Int] {
  val rand = new java.util.Random
  def generate = rand.nextInt()
}
```

Generators

Let's define a trait `Generator[T]` that generates random values of type `T`:

```
trait Generator[+T] {
  def generate: T
}
```

Some instances:

```
val booleans = new Generator[Boolean] {
  def generate = integers.generate > 0
}
```

Generators

Let's define a trait `Generator[T]` that generates random values of type `T`:

```
trait Generator[+T] {
  def generate: T
}
```

Some instances:

```
val pairs = new Generator[(Int, Int)] {
  def generate = (integers.generate, integers.generate)
}
```

Streamlining It

Can we avoid the `new Generator ...` boilerplate?

Ideally, we would like to write:

```
val booleans = for (x <- integers) yield x > 0

def pairs[T, U](t: Generator[T], u: Generator[U]) = for {
  x <- t
  y <- u
} yield (x, y)
```

What does this expand to?

Streamlining It

Can we avoid the `new Generator ...` boilerplate?

Ideally, we would like to write:

```scala
val booleans = integers map (x => x > 0)

def pairs[T, U](t: Generator[T], u: Generator[U]) =
  t flatMap (x => u map (y => (x, y)))
```

Need `map` and `flatMap` for that!

Generator with map and flatMap

Here’s a more convenient version of Generator:

```
trait Generator[+T] {
  self =>       // an alias for ”this”.

  def generate: T

  def map[S](f: T => S): Generator[S] = new Generator[S] {
    def generate = f(self.generate)
  }
```

Generator with map and flatMap

Here’s a more convenient version of Generator:

```
trait Generator[+T] {
  self =>       // an alias for ”this”.

  def generate: T

  def map[S](f: T => S): Generator[S] = new Generator[S] {
    def generate = f(self.generate)
  }

  def flatMap[S](f: T => Generator[S]): Generator[S] = new Generator[S] {
    def generate = f(self.generate).generate
  }
}
```

The booleans Generator

What does this definition resolve to?

```
val booleans = for (x <- integers) yield x > 0
```

The booleans Generator

What does this definition resolve to?

```
val booleans = for (x <- integers) yield x > 0
```

```
val booleans = integers map { x => x > 0 }
```

The booleans Generator

What does this definition resolve to?

```
val booleans = for (x <- integers) yield x > 0
```

```
val booleans = integers map { x => x > 0 }
```

```
val booleans = new Generator[Boolean] {
  def generate = (x: Int => x > 0)(integers.generate)
}
```

The booleans Generator

What does this definition resolve to?

```
val booleans = for (x <- integers) yield x > 0
```

```
val booleans = integers map { x => x > 0 }
```

```
val booleans = new Generator[Boolean] {
  def generate = (x: Int => x > 0)(integers.generate)
}
```

```
val booleans = new Generator[Boolean] {
  def generate = integers.generate > 0
}
```

The pairs Generator

```
def pairs[T, U](t: Generator[T], u: Generator[U]) = t flatMap {
  x => u map { y => (x, y) } }
```

The pairs Generator

```
def pairs[T, U](t: Generator[T], u: Generator[U]) = t flatMap {
  x => u map { y => (x, y) } }

def pairs[T, U](t: Generator[T], u: Generator[U]) = t flatMap {
  x => new Generator[(T, U)] { def generate = (x, u.generate) } }
```

The pairs Generator

```
def pairs[T, U](t: Generator[T], u: Generator[U]) = t flatMap {
  x => u map { y => (x, y) } }

def pairs[T, U](t: Generator[T], u: Generator[U]) = t flatMap {
  x => new Generator[(T, U)] { def generate = (x, u.generate) } }

def pairs[T, U](t: Generator[T], u: Generator[U]) = new Generator[(T, U)] {
  def generate = (new Generator[(T, U)] {
    def generate = (t.generate, u.generate)
  }).generate }
```

The pairs Generator

```
def pairs[T, U](t: Generator[T], u: Generator[U]) = t flatMap {
  x => u map { y => (x, y) } }

def pairs[T, U](t: Generator[T], u: Generator[U]) = t flatMap {
  x => new Generator[(T, U)] { def generate = (x, u.generate) } }

def pairs[T, U](t: Generator[T], u: Generator[U]) = new Generator[(T, U)] {
  def generate = (new Generator[(T, U)] {
    def generate = (t.generate, u.generate)
  }).generate }

def pairs[T, U](t: Generator[T], u: Generator[U]) = new Generator[(T, U)] {
  def generate = (t.generate, u.generate)
}
```

Generator Examples

```scala
def single[T](x: T): Generator[T] = new Generator[T] {
  def generate = x
}

def choose(lo: Int, hi: Int): Generator[Int] =
  for (x <- integers) yield lo + x % (hi - lo)

def oneOf[T](xs: T*): Generator[T] =
  for (idx <- choose(0, xs.length)) yield xs(idx)
```

A `List` Generator

A list is either an empty list or a non-empty list.

```
def lists: Generator[List[Int]] = for {
  isEmpty <- booleans
  list <- if (isEmpty) emptyLists else nonEmptyLists
} yield list
```

A List Generator

A list is either an empty list or a non-empty list.

```
def lists: Generator[List[Int]] = for {
  isEmpty <- booleans
  list <- if (isEmpty) emptyLists else nonEmptyLists
} yield list

def emptyLists = single(Nil)
```

A List Generator

A list is either an empty list or a non-empty list.

```
def lists: Generator[List[Int]] = for {
  isEmpty <- booleans
  list <- if (isEmpty) emptyLists else nonEmptyLists
} yield list

def emptyLists = single(Nil)

def nonEmptyLists = for {
  head <- integers
  tail <- lists
} yield head :: tail
```

A `Tree` Generator

Can you implement a generator that creates random `Tree` objects?

```scala
trait Tree

case class Inner(left: Tree, right: Tree) extends Tree

case class Leaf(x: Int) extends Tree
```

Hint: a tree is either a leaf or an inner node.

Application: Random Testing

You know about units tests:

- Come up with some some test inputs to program functions and a *postcondition*.
- The postcondition is a property of the expected result.
- Verify that the program satisfies the postcondition.

Question: Can we do without the test inputs?

Yes, by generating random test inputs.

Random Test Function

Using generators, we can write a random test function:

```
def test[T](g: Generator[T], numTimes: Int = 100)
    (test: T => Boolean): Unit = {
  for (i <- 0 until numTimes) {
    val value = g.generate
    assert(test(value), ”test failed for ”+value)
  }
  println(”passed ”+numTimes+” tests”)
}
```

Random Test Function

Example usage:

```
test(pairs(lists, lists)) {
  case (xs, ys) => (xs ++ ys).length > xs.length
}
```

Question: Does the above property always hold?

○ Yes

○ No

ScalaCheck

Shift in viewpoint: Instead of writing tests, write *properties* that are assumed to hold.

This idea is implemented in the `ScalaCheck` tool.

```
forAll { (l1: List[Int], l2: List[Int]) =>
  l1.size + l2.size == (l1 ++ l2).size
}
```

It can be used either stand-alone or as part of ScalaTest.

See ScalaCheck tutorial on the course page.