Infinite data in Scala

Mark Armstrong

October 6, 2020

Contents

1 Introduction

These notes were created for, and in some parts **during**, the lecture on October 2nd and the following tutorials.

2 Motivation

In our current lectures, we have been discussing the λ-calculus, and as of the time of writing this, we are about to discuss reduction strategies.

The reduction strategies used in the λ-calculus correspond to the parameter-passing methods used in conventional programming languages.

3 Call by value and call by name

Today, we are interested in the "call by value" and "call by name" reduction strategies/parameter-passing methods.

- Under "call by value" semantics, a function application is only evaluated after its arguments have been reduced to values.
- Under "call by name" (or "call by need") semantics, no arguments in a function application are evaluated before the function is applied.
  - (And no reduction is allowed inside abstractions.)

(Under both schemes, the leftmost, outermost valid reduction is done first.)

We can see the difference by considering a sample redex in the λ-calculus.

```
(λ x → x x)((λ y → y) (λ z → z))
```

Notice that there are two possible applications to carry out; applying `λ z → z` to `λ y → y`, or applying `(λ y → y) (λ z → z)` to `λ x → x x`.

A call-by-value semantics requires that the right side be evaluated first; "a function application is only evaluated after its arguments have been reduced to values". So we cannot perform the outermost application until the term on the right is reduced.

```
  (λ x → x x)((λ y → y) (λ z → z))
⟶ (λ x → x x)(λ z → z)
⟶ (λ z → z) (λ z → z)
→ λ z → z
= id
```

A call-by-name semantics instead requires that the outside application is evaluated first; "no arguments in a function application are evaluated before the function is applied". So we cannot perform the application in the term on the right until we apply the outermost application.

```
  (λ x → x x)((λ y → y) (λ z → z))
⟶ ((λ y → y) (λ z → z)) ((λ y → y) (λ z → z))
→ (λ z → z) ((λ y → y) (λ z → z))
→ (λ y → y) (λ z → z)
→ λ z → z
= id
```

4 Parameter passing in Scala

By default, Scala uses a call-by-value strategy.

```scala
def f(x: Int): Unit = { println("Called f with argument");
 ↪ println(x) }
```

You may *opt-in* to call by name using what they call “by name parameters”; simply prepend `=>` to the type.

```scala
def g(y: => Int) : Unit = { println("Called g with argument");
 ↪ println(y) }
```

Let us create a value that is not immediately evaluated, and which communicates when it is evaluated, so we can use it as an argument to test out our above definitions. One way to do this is by defining it as a method with no parameters.

```scala
def x: Int = { println("Evaluated x"); 1 }
```

:TODO: explain the term lazy

Scala also has “lazy” values, which are not evaluated until used. :TODO: how is this different than x?

```scala
lazy val y: Int = { println("Evaluated y"); 2 }
```

5 What’s the point?

Why do we want to be allowed to use call by name semantics and lazy values?

It may make some tasks easier conceptually; for instance, one common use case involves a function on the natural numbers, such as one that returns the “nth” prime.

We could certainly write such a function, but an alternative approach is to *lazily* construct the list of all primes by *filtering* the list of all naturals.

Since it is lazily constructed, no space is used until we begin to look up elements in the list. (Code courtesy of this StackOverflow post, modified to work with `LazyList`.)

```
lazy val ps: LazyList[Int] =
  2 #:: LazyList.from(3).filter(i => ps.takeWhile{j => j * j
   ↪  <= i}.forall{ k => i % k > 0});
```

And, because lazy data is not re-evaluated, once we have looked up elements, the already calculated portion of the list is automatically cached for us! This is (in some instances) an advantage over the function.

Now, we have subtlely used another concept enabled by lazy values and call-by-name semantics in the above: an infinite list!

6 Infinite data!

6.1 Not with lists...

When we discuss lists in computer science, they are usually defined as having *finite* length.

Let us try to break away from that convention. We can define an infinite list by using recursion. But what does you intuition tell you will happen here?

```
lazy val ones: List[Int] = 1 :: ones
```

6.2 ...instead, with lazy lists or streams!

The lazy list type and the stream type (now deprecated in favour of lazy lists) actually allow us to define such lists.

```
lazy val ones: LazyList[Int] = 1 #:: ones
```

We can then make these lists more interesting by filtering, zipping, etc.; or by writing more interesting recursive definitions.

7 ~~(Not)~~ Implementing our own infinite datatypes

7.1 The approach

To implement such a type `T` with the algebraic datatype approach we have been using, we may use (recursive) parameters of the form `Unit => T`; that is, the type of *functions* from `Unit` to `T`. (Recall that `Unit` is the type with just one value, written `()` in Scala.)

Function parameters are *never* evaluated until the function is invoked (called.) **This will approach will work in any language with higher-order functions.**

For example, we can define our own variant of streams.

```
sealed trait Stream[+A] // Stream is covariant (marked by the
 ↪  +)
case object SNil extends Stream[Nothing] // The singleton Nil
 ↪  object
case class Cons[A](a: A, f: Unit => Stream[A]) extends
 ↪  Stream[A]
```

7.2 Dealing with the tail function

Given a stream of the form `Cons(a,f)`, the `f` parameter does not directly give us the "tail" stream; we must invoke it, writing `f()`, to obtain the tail.

In the other direction, in order to construct a stream of the form `Cons(a,f)`, we must "wrap" the tail stream in a function definition. Rather than writing out a `defun` and giving this function a name (which in Scala lingo would make it a method), we can use an *anonymous* definition, such as below.

```
val ones: Stream[Int] = Cons(1, _ => ones)
```

The `_ => ones` indicates a function of one argument; the `_` is used in place of a variable name, since we are not using the variable (because it is of type `Unit`, there is really no use in giving it a name; it must be `()`.)

So, to construct a stream, we must "wrap" the tail; this is also called **delaying** the tail.
To deconstruct (stream) a stream, we must "invoke" the tail; this is also called **forcing** the tail.
The concepts of delaying and forcing data are generally applicable; we may later even consider a type which is just (potentially) delayed data.

7.3 `case object`?

We adopt here and now the convention of making our base case a `case object`, rather than a `case class`. The difference is that there is exactly one instance of a `case object` (it is a singleton), whereas there can be many of a `case class`. Since there can only be one instance of `SNil`, this instance needs to be a member of `Stream[A]` for any `A`, so it must be a member of `Stream[Nothing]`, since `Nothing` is the only subtype of all types. This does require us to mark `Stream` as *covariant*, meaning that `Stream[A]` is a subtype of `Stream[B]` if `A` is a subtype of `B`. (We will discuss subtyping, variance and covariance in more detail later in the course.)

7.4 Defining some methods on our streams

We need some methods to allow us to do even basic tasks with our new Stream type.

For instance, we need some convenient ways to define streams. For instance, we can construct constant streams that just repeat the same value.

```
def constantStream[A](a: A): Stream[A] =
  Cons(a, _ => constantStream(a))
```

But it's relatively hard to work with streams when we can only see the first value when they are printed. So, we pause giving ways to construct streams in order to give a way to *deconstruct* them. This method converts the first number of elements into a list which can be nicely and easily displayed.

```
def take[A](n: Int, s: => Stream[A]): List[A] = s match {
  case SNil => Nil
  case Cons(a,f) => n match {
    case n if n > 0 => a :: take(n-1,f())
    case _ => Nil
    }
  }
```

Now, let's define a method that will make it easier to define some other ways of constructing streams: prepending a finite number of elements from a list to a stream.

```
def prepend[A](l: List[A], s: => Stream[A]): Stream[A] = l
 ↪  match {
    case Nil => s
    case (h :: t) => Cons(h, _ => prepend(t, s))
  }
```

7.5 Take care: make arguments of type `Stream` by name arguments

Notice that in the definition of `prepend`, which is our first method to take a stream as an argument, we marked the stream as a "by name" argument, making it lazily evaluated.

This is because if we try to use `prepend` to define a stream *recursively*, so the recursive call is inserted as an argument to `prepend` (e.g., `val ones: Stream[Int] = repeat(List(1), ones)`), if that argument were *not* by name, it would be evaluated, and we would get a stack overflow due to unbounded recursion (this occurred during the lecture on Monday, October 5th.)

In general, mark your arguments of type `Stream` by name, in order to avoid accidental evaluation.

7.6 More methods on our streams

Using `prepend`, let us define two ways to go from a stream to a list; either just convert the list to a finite stream (one that actually does end with `SNil` eventually), or "repeat" the list infinitely.

```
def toStream[A](l: List[A]): Stream[A] = prepend(l, SNil)

def repeat[A](l: List[A]): Stream[A] = prepend(l, repeat(l))
```

We can also *append* a list to a stream, or more appropriately, append one stream to another. You might ask here though, "what happens if the stream is infinite"? The answer: nothing! The append will just never take place, since we never reach the `SNil` case while moving though the first stream.

```
def append[A](s: => Stream[A], t: => Stream[A]): Stream[A] = s
 ↪ match {
    case SNil => t
    case Cons(a, f) => Cons(a, _ => append(f(),t))
  }
```