Infinite data in Scala

Mark Armstrong

October 2, 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; specifically:

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 a "call by value" strategy, a function application is only evaluated after its arguments have been reduced to values.
- Under a "call by name" (or "call by need") strategy, 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 `y` to `λ x → x`, or applying `(λ x → x) y` to `λ x → x x`.

A call-by-value strategy 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 outmost 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-need strategy 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.

```
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.

```
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.

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

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

```
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 (now deprecated in favour of lazy lists) Stream type 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 Implementing our own infinite datatypes

We certainly can implement our own types of infinite data in Scala.

Unfortunately, we cannot do so using the "algebraic datatype" approach we have been using. And at least in this exercise, we are not going to investigate how to construct these types in Scala.

In other functional languages, in particular Haskell, which uses call-by-need semantics (a more performant take on call-by-name), defining your own infinite datatypes is simpler.