CSci 555, Functional Programming, Spring 2016
Functional Programming in Scala
Strictness and Laziness

H. Conrad Cunningham

14 April 2016 (minor formatting updates 3 August 2016)

Contents

Acknowledgements: This is a set of notes written to accompany my lectures on Chapter 5 of the book *Functional Programming in Scala* by Paul Chiusano and Runar Bjarnason (Manning, 2015). I constructed the notes around the ideas and Scala examples from that chapter and its accompanying materials. I encourage all students to study the chapter.

Prerequisite: This discussion assumes the reader is familiar with the programming concepts and Scala features covered in my *Notes on Scala for Java Programmers*, *Recursion Concepts and Terminology*, *Functional Data Structures*, and *Handling Errors without Exceptions*.

Advisory: The HTML version of this document uses MathML in a few locations. For best results, use a browser that supports the display of MathML. A good choice as of April 2016 seems to be a recent version of Firefox from Mozilla.

Strictness and Laziness

Introduction

The big idea we discuss in this chapter is how we can exploit nonstrict functions to increase efficiency, increase code reuse, and improve modularity in functional programs.

Motivation

In our discussion of Chapter 3 of *Functional Programming in Scala*, we examined purely functional data structures–in particular, the immutable, singly linked list.

We also examined the design and use of several bulk operations–such as `map`, `filter`, `foldLeft`, and `foldRight`. Each of these operations makes a pass over the input list and often constructs a new list for its output.

Consider the Scala expression

```
List(10,20,30,40,50).map(_ / 10).filter(_ % 2 == 1).map(_ * 100)
```

that generates the result:

```
List(100, 300, 500)
```

The evaluation of the expression requires three passes through the list. However, we could code a specialized function that does the same work in one pass.

```
def mfm(xs: List[Int]): List[Int] = xs match {
  case Nil     => Nil
  case (y::ys) =>
    val z = y / 10
    if (z % 2 == 1) (z*100) :: mfm(ys) else mfm(ys)
}
```

It would be convenient if we could instead get a result similar to `mfm` by composing simpler functions like `map` and `filter`.

Can we do this?

We can by taking advantage of *nonstrict* functions to build a *lazy* list structure. We introduced the concepts of strict and nonstrict functions in Chapter 4 and elaborate on them in this chapter.

What is strictness and nonstrictness?

If the evaluation of an expression runs forever or throws an exception instead of returning an explicit value, we say the expression does not *terminate*–or that it evaluates to *bottom* (written symbolically as ⊥).

A function `f` is *strict* if `f(x)` evaluates to bottom for all `x` that themselves evaluate to bottom. That is, `f(⊥) == ⊥`. A strict function's argument must always have a value for the function to have a value.

A function is *nonstrict* (sometimes called *lenient*) if it is not strict. That is, `f(⊥) != ⊥`. The function can sometimes have value even if its argument does not have a value.

For multiparameter functions, we sometimes apply these terms to individual parameters. A *strict* parameter of a function must always be evaluated by the function. A *nonstrict* parameter of a function may sometimes be evaluated by the function and sometimes not.

Exploring nonstrictness

By default, Scala functions are strict.

However, some operations are nonstrict. For example, the "short-circuited" `&&` operation is nonstrict; it does not evaluate its second operand when the first operation is `false`. Similarly, `||` does not evaluate its second operand when its first operand is `true`.

Consider the `if` expression as a ternary operator. When the condition operand evaluates to `true`, the operator evaluates the second (i.e., then) operand but not the third (i.e., else) operand. Similarly, when the condition is `false`, the operator evaluates the third operand but not the second.

We could implement `if` as a function as follows:

```
def if2[A](cond: Boolean, onTrue: () => A, onFalse: () => A): A =
  if (cond) onTrue() else onFalse()
```

Then we can call `if2` as in the code fragment

```
val age = 21
if2(age >= 18, () => println("Can vote"),() => println("Cannot vote"))
```

and get the output

```
Can vote
```

The parameter type `() => A` means that the corresponding argument is passed as a parameterless function that returns a value of type `A`. This function wraps the expression, which is not evaluated before the call. This function is an explicitly specified *thunk*.

When the value is needed, then the called function must *force* the evaluation of the thunk by calling it explicitly, for example by using `onTrue()`

To use the approach above, the caller must explicitly create the thunk. However, as we saw in the previous chapter, Scala provides *call-by-name* parameter passing that relieves the caller of this requirement in most circumstances. We can thus rewrite `if2` as follows:

```
def if2[A](cond: Boolean, onTrue: => A, onFalse: => A): A =
  if (cond) onTrue else onFalse
```

The `onTrue: => A` notation makes the argument expression a by-name parameter. Scala automatically creates the thunk for parameter `onTrue` and enables it to be referenced within the function without explicitly forcing its evaluation, for example by using `onTrue`.

An advantage of call-by-name parameter passing is that the evaluation of an expression can be delayed until its value is referenced, which may be never. A disadvantage is that the expression will be evaluated every time it is referenced.

To determine how to address this disadvantage, consider function

```
def maybeTwice(b: Boolean, i: => Int) = if (b) i + i else 0
```

which can be called as

```
println(maybeTwice(true, {println("hi"); 1 + 41}))
```

to generate output:

```
hi
hi
84
```

Note that the argument expression `i` is evaluated twice.

We can address this issue by defining a new variable and initializing it *lazily* to have the same value as the by-name parameter. We do this by declaring the temporary variable as a `lazy val`. The temporary variable will not be initialized until it is referenced, but it caches the calculated value so that it can be used without reevaluation on subsequent references.

We can rewrite `maybeTwice` as follows:

```
def maybeTwice2(b: Boolean, i: => Int) = {
  lazy val j = i
  if (b) j+j else 0
}
```

Now calling it as

```
println(maybeTwice2(true, {println("hi"); 1 + 41}))
```

generates output:

```
hi
84
```

This technique of caching the result of the evaluation gives us *call-by-need* parameter passing as it is called in Haskell and other lazily evaluated languages.

Lazy Lists

Now let's return to the problem discussed in the Motivation subsection. How can we use laziness to improve efficiency and modularity of our programs?

In this section, we answer this question by developing *lazy lists* or *streams*. These allow us to carry out multiple operations on a list without always making multiple passes over the elements.

Consider a simple algebraic data tupe `Stream`. A nonempty stream consists of a head and a tail, both of which must be nonstrict.

For technical reasons, Scala does not allow by-name parameters in the constructors for case classes. Thus these components must be explicitly defined thunks whose evaluations are explicitly forced when their values are required.

```
import scala.{Stream => _, _}
import Stream._

sealed trait Stream[+A]
case object Empty extends Stream[Nothing]
case class Cons[+A](h: () => A, t: () => Stream[A]) extends Stream[A]

object Stream {
  def cons[A](hd: => A, tl: => Stream[A]): Stream[A] = {
    lazy val head = hd            // cache values once computed
    lazy val tail = tl
    Cons(() => head, () => tail)  // create thunks for Cons
  }
  def empty[A]: Stream[A] = Empty
  def apply[A](as: A*): Stream[A] =
    if (as.isEmpty) empty else cons(as.head, apply(as.tail: _*))
}
```

Memoizing streams

In the `Stream` data type, we define two *smart constructors* to create new streams. By convention, these are functions defined in the companion object with the same names as the corresponding type constructors except they begin with a lowercase letter. They construct a data type object, ensuring that the needed integrity invariants are established. In the `Stream` type, these take care of the routine work of creating the thunks, caching the values, and enabling transparent use of the parameters.

Smart constructor function `cons` takes the head and tail of the new `Stream` as by-name parameters, equates these to lazy variables to cache their values, and then creates a `Cons` cell whose fields are two explicit thunks wrapping the head and the tail.

Smart constructor function `empty` just creates an `Empty` `Stream`.

Both smart constructors have return type `Stream[A]`. In addition to establishing the needed invariants, the use of the smart constructors helps Scala's type inference mechanism infer the `Stream` type (which is what we usually want) instead of the subtypes associated with the case class/object constructors (which is what often will be inferred in Scala's object-oriented type system).

Convenience function `apply` takes a sequence of zero or more arguments and creates the corresponding `Stream`.

If a function examines or traverses a `Stream`, it must explicitly force the thunks. In general, we should encapsulate such accesses within functions defined as a part of the `Stream` implementation. (That is, we should practice *information hiding*, hide this design detail as a *secret* of the `Stream` implementation as we discuss in the notes on Data Abstraction.)

An example of this is function `headOption` that optionally extracts the head of the stream.

```
def headOption: Option[A] = this match {
  case Empty     => None
  case Cons(h,t) => Some(h()) // force thunk
}
```

It explicitly forces the thunk and thus enables code that called it to work with the values.

This technique for caching the value of the by-name argument is an example of memoizing the function. In general, *memoization* is an implementation technique in which a function stores the return value computed for certain arguments. Instead of recomputing the value on a subsequent call, the function just returns the cached values. This technique uses memory space to (potentially) save computation time later.

Helper functions

Now let's define a few functions that help us manipulate streams. We implement these as methods on the `Stream` trait.

First, let's define a function `toList` that takes a `Stream` (as its implicit argument) and constructs the corresponding Scala `List`. A standard backward recursive method can be defined as follows:

```
def toListRecursive: List[A] = this match {
  case Cons(h,t) => h() :: t().toListRecursive // force thunks
  case _         => List()
}
```

Of course, this method may suffer from stack overflow for large streams. We can remedy this by using a tail recursive auxiliary function that uses an accumulator to build up the list in reverse order and then reverses the constructed list.

```
def toList: List[A] = {
  @annotation.tailrec
  def go(s: Stream[A], acc: List[A]): List[A] = s match {
    case Cons(h,t) => go(t(), h() :: acc) // force thunks
    case _         => acc
  }
  go(this, List()).reverse
}
```

To avoid the `reverse`, we could instead build up the list in a mutable `ListBuffer` using a loop and then, when finished, convert the buffer to an immutable `List`. We preserve the *purity* of the `toList` function by encapsulating use of the mutable buffer inside the function.

```
def toListFast: List[A] = {
  val buf = new collection.mutable.ListBuffer[A]
  @annotation.tailrec
  def go(s: Stream[A]): List[A] = s match {
    case Cons(h,t) =>
      buf += h() // force head thunk, add to end of buffer
      go(t())    // force tail thunk, process recursively
    case _ => buf.toList  // convert buffer to immutable list
  }
  go(this)
}
```

Next, let's define function `take` to return the first `n` elements from a `Stream` and function `drop` to skip the first `n` elements.

We can define method `take` using a standard backward recursive form that matches on the structure of the implicit argument. However, we must be careful not to evaluate either the head or the tail thunks unnecessarily (e.g., by treating the `n == 1` and `n == 0` cases specially).

```
def take(n: Int): Stream[A] = this match {
  case Cons(h, t) if n > 1  => cons(h(), t().take(n - 1))
  case Cons(h, _) if n == 1 => cons(h(), empty)
```

< case _ => empty // stream empty or n < 1 }

We can define method `drop` to recursively calling `drop` on the forced tail. This yields the following tail recursive function.

```
@annotation.tailrec
final def drop(n: Int): Stream[A] = this match {
  case Cons(_, t) if n > 0 => t().drop(n - 1)
  case _ => this
}
```

Finally, let's also define method `takeWhile` to return all starting elements of the `Stream` that satisfy the given predicate.

```
def takeWhile(p: A => Boolean): Stream[A] = this match {
  case Cons(h,t) if p(h()) => cons(h(), t() takeWhile p)
  case _ => empty
}
```

In the first case, we apply method `takeWhile` as an infix operator.

Separating Program Description from Evaluation

One of the fundamental design concepts in software engineering and programming is *separation of concerns*. A concern is some set of information that affects a software system. We identify the key concerns in a software design and try to keep them separate and independent from each other. The goal is to implement the parts independently and then combine the parts to form a complete solution.

We apply separation of concerns in modular programming and abstract data types as *information hiding*. We hide the *secrets* of how a module is implemented (e.g., what algorithms and data structures are used, what specific operating system or hardware devices are used, etc.) from the external users of the module or data type. We encapsulate the secrets behind an *abstract interface*.

We also apply separation of concerns in software architecture for computing applications. For example, we try to keep the *business logic* (i.e., specific knowledge about the application area), the user interface, and the data representation for an application separate from each other using an approach such as the *Model-View-Controller* (MVC) architectural design pattern.

In functional programming, we also apply separation of concerns by seeking to *keep the description of computations separate from their evaluation* (execution). Examples include:

- first-class functions that express computations in their bodies but which must be supplied arguments before they execute
- use of `Option` or `Either` to express that an error has occurred but deferring the handling of the error to other parts of the program

- use of `Stream` operators to assemble a computation that generates a sequence without actually running the computation until later when its result in needed

Laziness promotes reuse

In general, lazy evaluation enables us to separate the description of an expression from the evaluation of the expression. It enables us to to describe a "larger" expression than we need and then to only evaluate the portion that we actually need. This offers us the potential for greater *code reuse*.

Consider a method `exists` on `Stream` that checks whether an element matching a Boolean function `p` occurs in the stream. We can define this using an explicit tail recursion as follows:

```
def exists(p: A => Boolean): Boolean = this match {
  case Cons(h,t) => p(h()) || t().exists(p)
  case _         => false
}
```

Given that `||` is nonstrict in its second argument, this function terminates and returns `true` as soon as it finds the first element that makes `p` true. Because the stream holds the tail in a `lazy val`, it is only evaluated when needed. So `exists` does not evaluate the stream past the first occurrence.

As with the `List` data type in Chapter 3, we can define a more general method `foldRight` on `Stream` to represent the pattern of computation exhibited by `exists`.

```
def foldRight[B](z: => B)(f: (A, => B) => B): B = this match {
  case Cons(h,t) => f(h(), t().foldRight(z)(f))
  case _         => z
}
```

The notation `=> B` means that combining function `f` takes its second argument by-name and, hence, may not evaluate it in all circumstances. If `f` does not evaluate its second argument, then the recursion terminates. Thus the overall `foldRight` computation can terminate before it completes the complete traversal through the stream.

We can now redefine `exists` to use the more general function as follows:

```
def exists2(p: A => Boolean): Boolean =
  foldRight(false)((a, b) => p(a) || b)
```

Here parameter `b` represents the unevaluated recursive step that folds the tail of the stream. If `p(a)` returns `true`, then `b` is not evaluated and the computation terminates early.

Caveat: The second version of `exists` illustrates how we can use a general function to represent a variety of more specific computations. But, for a large stream in which all elements evaluate to `false`, this version is not stack safe.

Because the `foldRight` method on `Stream` can terminate its traversal early, we can use it to implement `exists` efficiently. Unfortunately, we cannot implement the `List` version of `exists` efficiently in terms of the `List` version of `foldRight`. We must implement a specialized recursive version of `exists` to get early termination.

Laziness thus enhances our ability to reuse code.

Incremental computations

Now, let's flesh out the `Stream` trait and implement the basic `map`, `filter`, `append`, and `flatMap` methods using the general function `foldRight`, as follows:

```
def map[B](f: A => B): Stream[B] =
  foldRight(empty[B])((h,t) => cons(f(h), t))

def filter(p: A => Boolean): Stream[A] =
  foldRight(empty[A])((h,t) => if (p(h)) cons(h, t) else t)

def append[B >: A](s: => Stream[B]): Stream[B] =
  foldRight(s)((h,t) => cons(h,t))

def flatMap[B](f: A => Stream[B]): Stream[B] =
  foldRight(empty[B])((h,t) => f(h) append t)
```

These implementations are *incremental*. They do not fully generate all their answers. No computation takes place until some other computation examines the elements of the output `Stream` and then only enough elements are generated to give the requested result.

Because of their incremental nature, we can call these functions one after another without fully generating the intermediate results.

We can now address the problem raised by the problem in the Introduction to these notes. There we asked the question of how can we compute the result of the expression

```
List(10,20,30,40,50).map(_ / 10).filter(_ % 2 == 1).map(_ * 100)
```

without producing two unneeded intermediate lists.

The `Stream` expression

```
Stream(10,20,30,40,50).map(_ / 10).filter(_ % 2 == 1).map(_ * 100).toList
```

generates the result

```
List(100, 300, 500)
```

which is the same as the `List` expression. The expression looks the same except that we create a `Stream` initially instead of a `List` and we call `toList` to force evaluation of stream at the end.

When executed, the lazy evaluation interleaves two `map`, the `filter`, and the `toList` transformations. The computation does not fully instantiate any intermediate streams. It is a similar interleaving to what we did in the special purpose function `mfm` in the introduction.

(For a more detailed discussion of this interleaving, see Listing 5.3 in the *Functional Programming in Scala* book.)

Because stream computations do not generate intermediate streams in full, we are free to use stream operations in ways that might seem counterintuitive at first. For example, we can use `filter` (which seems to process the entire stream) to implement `find`, a function to return the first occurrence of an element in a stream that satisfies a given predicate, as follows:

```
def find(p: A => Boolean): Option[A] = filter(p).headOption
```

The incremental nature of these computations can sometimes save memory. The computation may only need a small amount of working memory; the garbage collector can quickly recover working memory that the current step does not need.

Of course, some computations may require more intermediate elements and each element may itself require a large amount of memory, so not all computations are as well-behaved as the examples in this section.

For comprehensions on streams

Given that we have defined `map`, `filter`, and `flatMap`, we can now use sequence comprehensions on our `Stream` data. For example, the code fragment

```
val seq = for (x <- Stream(1,2,3,4) if x > 2; y <- Stream(1,2)) yield x
println(seq.toList)
```

causes the following to print on the console:

```
List(3, 3, 4, 4)
```

Note: During compilation, the Scala compiler issues a deprecation warning that `filter` is used instead of `withFilter`. In a future release of Scala, this substitution may no longer work. Because `filter` is lazy for streams, we could define `withFilter` as an alias for `withFilter` with the following:

```
def withFilter = filter _
```

However, `filter` does generate a new `Stream` where `withFilter` normally does not generate a new collection. Although this gets rid of the warning, it would be better to implement a proper `withFilter` function.

Infinite Streams snd Corecursion

Because the streams are incremental, the functions we have defined also work for *infinite streams*.

Consider the following definition for an infinite sequence of ones:

```
lazy val ones: Stream[Int] = cons(1, ones)
```

Note: The book *Functional Programming in Scala* does not add the `lazy` annotation, but that version gives a compilation error. Adding `lazy` seemed to fix the problem, but this issue should be investigated further.

Although `ones` is infinite, the `Stream` functions only reference the finite prefix of the stream needed to compute the needed result.

For example:

- `ones.take(5).toList` yields `List(1,1,1,1,1)`
- `ones.map(_+2).take(5).toList` yields `List(3,3,3,3,3)`
- What about `ones.map(_+2).toList`?

We can generalize `ones` to a `constant` function as follows:

```
def constant[A](a: A): Stream[A] = {
  lazy val tail: Stream[A] = Cons(() => a, () => tail)
  tail
}
```

An alternative would be just to make the body `cons(a, constant(a))`. But the above is more efficient because it is just one object referencing itself.

We can also define an increasing `Stream` of all integers beginning with `n` as follows:

```
def from(n: Int): Stream[Int] =
  cons(n, from(n+1))
```

The (second-order) Fibonacci sequence begins with the elements 0 and 1; each subsequent element is the sum of the two previous elements. We can define the Fibonacci sequence as a stream `fibs` with the following definition:

```
val fibs = {
  def go(f0: Int, f1: Int): Stream[Int] =
    cons(f0, go(f1, f0+f1))
  go(0, 1)
}
```

Prime numbers: Sieve of Erastosthenes

A positive integer greater than 1 is *prime* if it is divisible only by itself and 1. The *Sieve of Eratosthenes* algorithm works by removing multiples of numbers once they are identified as prime.

- We begin the increasing stream of integers starting with 2, a prime number.
- The head is 2, so we remove all the multiples of 2 from the stream.
- The head of the tail is 3, so it is prime because it was not removed as a multiple of 2 and it is the smallest integer remaining.
- Continue the process recursively on the tail.

We can define this calculation with the following `Stream` functions.

```
def sieve(ints: Stream[Int]): Stream[Int] = ints.headOption match {
  case None => sys.error("Should not occur: No head on infinite stream.")
  case Some(x) => cons(x,sieve(ints drop 1 filter (_ % x > 0)))
}

val primes: Stream[Int] = sieve(from(2))
```

We can then use `primes` to define a function `isPrime` to test whether an integer is prime.

```
def isPrime(c: Int): Boolean =
  (primes filter (_ >= c) map (_ == c)).headOption getOrElse
    sys.error("Should not occur: No head on infinite list.")
```

Function `unfold`

Now let's consider `unfold`, a more general stream-building function. Function `unfold` takes an initial state and a function that produces both the next state and the next value in the stream. We can define it as follows:

```
def unfold[A, S](z: S)(f: S => Option[(A, S)]): Stream[A] =
  f(z) match {
    case Some((h,s)) => cons(h, unfold(s)(f))
    case None        => empty
  }
```

This function applies `f` to the current state `z` to generate the next state `s` and the next element `h` of the stream. We use `Option` so `f` can signal when to terminate the `Stream`.

Function `unfold` is an example of a corecursive function.

A *recursive* function consumes data. The input of each successive call is "smaller" than the previous one. Eventually the recursion terminates when input size reaches the minimum.

A *corecursive* function produces data. Corecursive functions need not terminate as long as they remain *productive*. By productive, we mean that the function can continue to evaluate more of the result in a finite amount of time.

The `unfold` function remains productive as long as its argument function `f` terminates. Function `f` must terminate for the `unfold` computation to reach its next state.

Some writers in the functional programming community use the term *guarded recursion* instead of corecursion and the term *cotermination* instead of productivity. See the Wikipedia articles on corecursion and coinduction for more information and links.

The function `unfold` is very general. For example, we can now define `ones`, `constant`, `from`, and `fibs` with `unfold`.

```
val onesViaUnfold = unfold(1)(_ => Some((1,1)))

def constantViaUnfold[A](a: A) =
  unfold(a)(_ => Some((a,a)))

def fromViaUnfold(n: Int) =
  unfold(n)(n => Some((n,n+1)))

val fibsViaUnfold =
  unfold((0,1)) { case (f0,f1) => Some((f0,(f1,f0+f1))) }
```

Summary

The big idea in this chapter is that we can exploit nonstrict functions to increase efficiency, increase code reuse, and improve the modularity in functional programs.