4 — Strictness and Laziness

Einführung in die Funktionale Programmierung

Prof. Frank Puppe, Felix Herrmann, Alexander Gehrke

Sommersemester 2020

Lehrstuhl für Informatik VI, Uni Würzburg

Laziness — Basics

Motivation

Consider the following code:

```
List(1,2,3,4)
  .map(_ + 10)
  .filter(_ % 2 == 0)
  .map(_ * 3)
```

Each call to `map` or `filter` creates a new intermediate list, which is then only used once for the next call:

```
List(1,2,3,4).map(_ + 10).filter(_ % 2 == 0).map(_ * 3)
List(11,12,13,14).filter(_ % 2 == 0).map(_ * 3)
List(12,14).map(_ * 3)
List(36,42)
```

Motivation

We’d like to fuse the transformations to run in one pass without abandoning the compositional style.

This can be achieved with non-strictness (commonly known as laziness).

Strictness and non-strictness

A function can be non-strict, meaning it can choose not to evaluate one or more of its arguments. Strict functions on the other hand always evaluate their arguments (default in most languages, many only support this).

Example: consider the (strict) method

```
def createDebugMessage(loglevel: Int, out: String): String =
  if loglevel >= 5 then
    "DEBUG: " + out
  else
    "Debug disabled"
```

- `createDebugMessage(3, "a" + "b")`
  will be passed the single string `"ab"`
- `createDebugMessage(3, expensiveMethod())`
  will call the expensive method, even if it doesn't use the result in the end
- `createDebugMessage(3, sys.error("failure"))`
  will not even enter function body, but throw immediately

Examples of non-strictness

You may already know the concept of non-strictness from other programming constructs:

- Short-circuiting `&&` and `||` operators
  e.g. in Java: `if(true || bar()) ...` will never call `bar`
- `if` can be viewed as a function with three parameters: condition, true-expression and false-expression, the latter two non-strict.
- Any language that allows passing callable code like functions around can achieve this effect, e.g. some Java logging libraries use this.

Non-strictness in Scala

Let’s look at that last two points, and implement `if` as a function taking functions:

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

val a = 4
if2(a < 22,
  () => println("a"),  // code blocks without parameters also called "thunk"
  () => println("b")
)
```

This is strict in its condition parameter, but we simulate non-strictness / laziness for the `ifTrue` and `ifFalse` arguments by making them parameterless functions (also called thunks). These will only be evaluated, if we call them explicitly.

Non-strictness in Scala

Scala has language support for making this nicer: prefix the type of an argument with `=>` to create a call-by-name parameter. This is equivalent to our previous implementation:

```
def if3[A](cond: Boolean, onTrue: => A, onFalse: => A): A =
  if cond then onTrue else onFalse

if3(a < 22, println("a"), println("b"))
```

As you can see, you do not need to pass functions explicitly and do not have to call them. But the `ifTrue` and `ifFalse` parameters are not evaluated until the code inside the method uses them.

Non-strict values

Non-strict function parameters behave just like parameterless functions, so using them multiple times also evaluates them multiple times (results are not cached):

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

scala> val x = maybeTwice(true, { println("hi"); 1+41 })
hi
hi
val x: Int = 84
```

Even if our expressions are pure and therefore should not behave differently if called twice, we wouldn’t want that for performance reasons.

Non-strict values

The solution: value declarations can be declared `lazy`, which means the right hand side is evaluated lazily, but only once on first access of the value:

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

scala> val x = maybeTwice2(true, { println("hi"); 1+41 })
hi
val x: Int = 84
```

Formal definition

Definition (Bottom)
If the evaluation of an expression runs forever, throws an error or otherwise breaks normal control flow instead of returning a definite value, we say the expression doesn't *terminate*, or that it evaluates to *bottom* ($\perp$) and its type is the bottom type (`Nothing`).

Definition (Strictness)
A function f is strict, if the expression $f(x)$ evaluates to bottom for all x that evaluate to bottom.

Informal explanation

So a function evaluates to bottom, if it never returns to where it was called from. As the definition says, examples for this include endless loops / recursions and exception throwing.

And a function is strict, when passing such a never-returning function as parameter always causes the expression including the function call to never return too (i.e. the passed parameter is always evaluated).

Alternatively, you can say a function f is strict, if all its parameters are evaluated exactly once *before* evaluating the function body.

Lazy Lists

Lazy Lists

Back to our motivational problem: we now have everything we need to create a lazily evaluated list:

```
enum LazyList[+A]:
  case Empty
  case Cons(h: () => A, t: () => LazyList[A])
```

This is almost identical to our `List`, except head and tail of our LazyList-Cons are given as thunks.

We have to use explicit thunks here, because call-by-name is not allowed for `val`s (such as case class parameters), and lazy is not allowed for parameters.

LazyList — Methods

To access the values, we have to force `h` explicitly via `h()`.

Example: implementation of `headOption` (return first element, if present) inside `LazyList` trait:

```
def headOption: Option[A] = this match
  case Cons(h, _) => Some(h())
  case Empty => None
```

LazyList — Methods

Values are not cached when using `Cons` directly. Calling `headOption` twice evaluates h( ) twice, which may be expensive.

Solution: provide *smart constructors*, which handle caching. This indirection also allows us to use by-name syntax.

```scala
object LazyList: // companion object
  def cons[A](h: => A, t: => LazyList[A]): LazyList[A] =
    lazy val head = h
    lazy val tail = t
    Cons(() => head, () => tail)

  def empty[A]: LazyList[A] = Empty

  def apply[A](as: A*): LazyList[A] =
    if as.isEmpty then empty
    else cons(as.head, apply(as.tail: _*))
```

LazyList — Methods

Values are not cached when using `Cons` directly. Calling `headOption` twice evaluates `h()` twice, which may be expensive.

Solution: provide *smart constructors*, which handle caching. This indirection also allows us to use by-name syntax.

```
object LazyList: // companion object
  def cons[A](h: => A, t: => LazyList[A]): LazyList[A] =
    lazy val head = h
    lazy val tail = t
    Cons(() => head, () => tail)

  def empty[A]: LazyList[A] = Empty

  def apply[A](as: A*): LazyList[A] =
    if as.isEmpty then empty
    else cons(as.head, apply(as.tail: _*))
```

Cache given values using lazy vals and wrap them in thunks for `Cons`

LazyList — Methods

Values are not cached when using `Cons` directly. Calling `headOption` twice evaluates `h()` twice, which may be expensive.

Solution: provide *smart constructors*, which handle caching. This indirection also allows us to use by-name syntax.

```
object LazyList: // companion object
  def cons[A](h: => A, t: => LazyList[A]): LazyList[A] =
    lazy val head = h
    lazy val tail = t
    Cons(() => head, () => tail)

  def empty[A]: LazyList[A] = Empty    Return empty lazy list with correct type

  def apply[A](as: A*): LazyList[A] =
    if as.isEmpty then empty
    else cons(as.head, apply(as.tail: _*))
```

LazyList — Methods

Values are not cached when using `Cons` directly. Calling `headOption` twice evaluates `h()` twice, which may be expensive.

Solution: provide *smart constructors*, which handle caching. This indirection also allows us to use by-name syntax.

```
object LazyList: // companion object
  def cons[A](h: => A, t: => LazyList[A]): LazyList[A] =
    lazy val head = h
    lazy val tail = t
    Cons(() => head, () => tail)

  def empty[A]: LazyList[A] = Empty

  def apply[A](as: A*): LazyList[A] =
    if as.isEmpty then empty
    else cons(as.head, apply(as.tail: _*))
```

Varargs constructor for passing several elements

↑ *we use the smart constructor here (cons instead of Cons)*

Exercise: Convert LazyList to List

Implement a method on `LazyList` that converts it to a `List`, i.e. forces its evaluation.

```scala
def toList: List[A]
```

You may use `List` from the standard library

Exercise: Convert LazyList to List — Solution

Natural recursive solution:

```scala
def toListRecursive: List[A] = this match
  case Cons(h,t) => h() :: t().toListRecursive
  case Empty => List()
```

Exercise: Convert LazyList to List — Solution

Tail-recursive solution:

```scala
def toList: List[A] =
  @annotation.tailrec
  def go(s: LazyList[A], acc: List[A]): List[A] = s match
    case Cons(h,t) => go(t(), h() :: acc)
    case Empty => acc
  go(this, List()).reverse
```

- Prepend each element to an accumulator
- Results in reversed list, call reverse at the end

Exercise: `take` and `drop`

Add the functions `take` for returning the first *n* elements of a lazy list, and `drop` for removing the first *n* elements to our `LazyList` trait:

```
def take(n: Int): LazyList[A]
```

```
def drop(n: Int): LazyList[A]
```

Exercise: take and drop — Solution

```
def take(n: Int): LazyList[A] = this match
  case Cons(h, t) =>
    if      n > 1  then cons(h(), t().take(n - 1))
    else if n == 1 then cons(h(), empty)
    else empty
  case Empty => empty
```

Note the use of lazy constructors to keep the lists lazy.

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

Common functions

We can implement the methods `map`, `filter`, `append` and `flatMap` we know from `List` for `LazyList` (which we will do on the exercise sheet).

Note that because of LazyList's lazy nature, a call to any of these does nothing, until the elements are actually requested.

LazyList method chaining

Let’s look step by step at a chained `map` and `filter`

```
LazyList(1,2,3,4).map(_ + 10).filter(_ % 2 == 0).toList
```

LazyList method chaining

Apply `map` to first element

```
LazyList(1,2,3,4).map(_ + 10).filter(_ % 2 == 0).toList
cons(11, LazyList(2,3,4).map(_ + 10)).filter(_ % 2 == 0).toList
```

LazyList method chaining

Apply `filter` to first element

```
LazyList(1,2,3,4).map(_ + 10).filter(_ % 2 == 0).toList
cons(11, LazyList(2,3,4).map(_ + 10)).filter(_ % 2 == 0).toList
LazyList(2,3,4).map(_ + 10).filter(_ % 2 == 0).toList
```

LazyList method chaining

Apply `map` to second element

```
LazyList(1,2,3,4).map(_ + 10).filter(_ % 2 == 0).toList
cons(11, LazyList(2,3,4).map(_ + 10)).filter(_ % 2 == 0).toList
LazyList(2,3,4).map(_ + 10).filter(_ % 2 == 0).toList
cons(12, LazyList(3,4).map(_ + 10)).filter(_ % 2 == 0).toList
```

LazyList method chaining

Apply `filter` to second element. Produce first result of list

```
LazyList(1,2,3,4).map(_ + 10).filter(_ % 2 == 0).toList
cons(11, LazyList(2,3,4).map(_ + 10)).filter(_ % 2 == 0).toList
LazyList(2,3,4).map(_ + 10).filter(_ % 2 == 0).toList
cons(12, LazyList(3,4).map(_ + 10)).filter(_ % 2 == 0).toList
12 :: LazyList(3,4).map(_ + 10).filter(_ % 2 == 0).toList
```

LazyList method chaining

And so on. . .

```
LazyList(1,2,3,4).map(_ + 10).filter(_ % 2 == 0).toList
cons(11, LazyList(2,3,4).map(_ + 10)).filter(_ % 2 == 0).toList
LazyList(2,3,4).map(_ + 10).filter(_ % 2 == 0).toList
cons(12, LazyList(3,4).map(_ + 10)).filter(_ % 2 == 0).toList
12 :: LazyList(3,4).map(_ + 10).filter(_ % 2 == 0).toList
12 :: cons(13, LazyList(4).map(_ + 10)).filter(_ % 2 == 0).toList
```

LazyList method chaining

And so on...

```
LazyList(1,2,3,4).map(_ + 10).filter(_ % 2 == 0).toList
cons(11, LazyList(2,3,4).map(_ + 10)).filter(_ % 2 == 0).toList
LazyList(2,3,4).map(_ + 10).filter(_ % 2 == 0).toList
cons(12, LazyList(3,4).map(_ + 10)).filter(_ % 2 == 0).toList
12 :: LazyList(3,4).map(_ + 10).filter(_ % 2 == 0).toList
12 :: cons(13, LazyList(4).map(_ + 10)).filter(_ % 2 == 0).toList
12 :: LazyList(4).map(_ + 10).filter(_ % 2 == 0).toList
```

LazyList method chaining

And so on. . .

```
LazyList(1,2,3,4).map(_ + 10).filter(_ % 2 == 0).toList
cons(11, LazyList(2,3,4).map(_ + 10)).filter(_ % 2 == 0).toList
LazyList(2,3,4).map(_ + 10).filter(_ % 2 == 0).toList
cons(12, LazyList(3,4).map(_ + 10)).filter(_ % 2 == 0).toList
12 :: LazyList(3,4).map(_ + 10).filter(_ % 2 == 0).toList
12 :: cons(13, LazyList(4).map(_ + 10)).filter(_ % 2 == 0).toList
12 :: LazyList(4).map(_ + 10).filter(_ % 2 == 0).toList
12 :: cons(14, LazyList[Int]().map(_ + 10)).filter(_ % 2 == 0).toList
```

LazyList method chaining

Apply `filter` to fourth element. Produce final result of list

```
LazyList(1,2,3,4).map(_ + 10).filter(_ % 2 == 0).toList
cons(11, LazyList(2,3,4).map(_ + 10)).filter(_ % 2 == 0).toList
LazyList(2,3,4).map(_ + 10).filter(_ % 2 == 0).toList
cons(12, LazyList(3,4).map(_ + 10)).filter(_ % 2 == 0).toList
12 :: LazyList(3,4).map(_ + 10).filter(_ % 2 == 0).toList
12 :: cons(13, LazyList(4).map(_ + 10)).filter(_ % 2 == 0).toList
12 :: LazyList(4).map(_ + 10).filter(_ % 2 == 0).toList
12 :: cons(14, LazyList[Int]().map(_ + 10)).filter(_ % 2 == 0).toList
12 :: 14 :: LazyList[Int]().map(_ + 10).filter(_ % 2 == 0).toList
```

LazyList method chaining

All mapped and filtered, convert empty lazy list to empty strict list

```
LazyList(1,2,3,4).map(_ + 10).filter(_ % 2 == 0).toList
cons(11, LazyList(2,3,4).map(_ + 10)).filter(_ % 2 == 0).toList
LazyList(2,3,4).map(_ + 10).filter(_ % 2 == 0).toList
cons(12, LazyList(3,4).map(_ + 10)).filter(_ % 2 == 0).toList
12 :: LazyList(3,4).map(_ + 10).filter(_ % 2 == 0).toList
12 :: cons(13, LazyList(4).map(_ + 10)).filter(_ % 2 == 0).toList
12 :: LazyList(4).map(_ + 10).filter(_ % 2 == 0).toList
12 :: cons(14, LazyList[Int]().map(_ + 10)).filter(_ % 2 == 0).toList
12 :: 14 :: LazyList[Int]().map(_ + 10).filter(_ % 2 == 0).toList
12 :: 14 :: List()
```

Without `toList` call, nothing would have been executed yet

Separating program description from evaluation

Laziness lets us separate the description of an expression from its evaluation. This in turn allows to describe larger expression than needed, but only evaluating parts of it.

```
def exists(p: A => Boolean): Boolean = this match
  case Cons(x, xs) => p(x()) || xs().exists(p)
  case Empty => false
```

The right hand side of `||` is non-strict ⇒ `xs()` is not called, if `p(x())` is true.

If it was strict, we'd need an if-else-construct here, this way we have a single boolean expression for a function returning a boolean result.

Folding and unfolding LazyLists

General Recursion for LazyList

We can implement `foldRight` like for lists, but with a lazy start value and lazy parameter in the passed function:

```scala
def foldRight[B](z: => B)(f: (A, => B) => B):B = this match
  case Cons(x, xs) => f(x(), xs().foldRight(z)(f))
  case Empty => z
```

If `f` does not evaluate its second parameter, the LazyList's tail and `z` are never evaluated $\Rightarrow$ allows early termination of loop!

General Recursion for LazyList

We can implement `foldRight` like for lists, but with a lazy start value and lazy parameter in the passed function:

```
def foldRight[B](z: => B)(f: (A, => B) => B):B = this match
  case Cons(x, xs) => f(x(), xs().foldRight(z)(f))
  case Empty => z
```

If `f` does not evaluate its second parameter, the LazyList's tail and `z` are never evaluated ⇒ allows early termination of loop! `exists` in terms of `foldRight`:

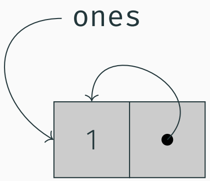
```
def existsFR(p: A => Boolean): Boolean =
  foldRight(false)((a,b) => p(a) || b)
```

If we swap `false` with `true` and `||` with `&&`, we get a `forall` function

Infinite LazyLists

Lazy evaluation allows our functions to work on infinite LazyLists, e.g.

```scala
val ones: LazyList[Int] = LazyList.cons(1, ones) // infinite ones
```

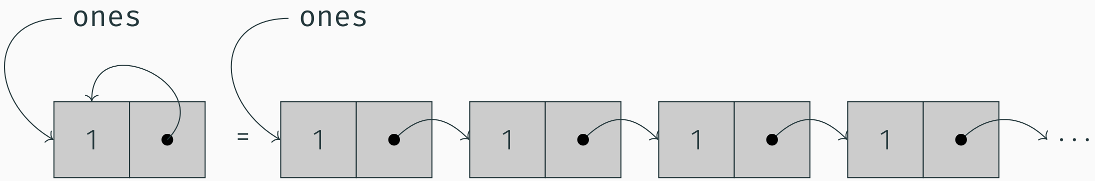


- `ones` is defined recursively. This only works, because the recursive use occurs in a non-strict position.

Infinite LazyLists

Lazy evaluation allows our functions to work on infinite LazyLists, e.g.

```scala
val ones: LazyList[Int] = LazyList.cons(1, ones) // infinite ones
```

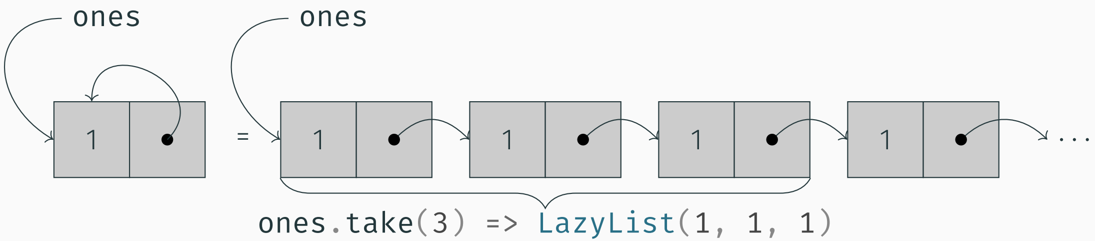


- `ones` is defined recursively. This only works, because the recursive use occurs in a non-strict position.
- This is equivalent to an infinite chain of cons cells.

Infinite LazyLists

Lazy evaluation allows our functions to work on infinite LazyLists, e.g.

```scala
val ones: LazyList[Int] = LazyList.cons(1, ones) // infinite ones
```



- `ones` is defined recursively. This only works, because the recursive use occurs in a non-strict position.
- This is equivalent to an infinite chain of cons cells.
- LazyList functions evaluate only the necessary part, e.g. `take(3)` only evaluates the first three cells.

Exercise: Fibonacci LazyList

Implement a function on the companion, that generates an infinite LazyList of Fibonacci numbers (the *n*-th Fibonacci number is the sum of the previous two, with 0 and 1 as starting values):

$$F_0 = 0$$

$$F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2}$$

Tip: use a recursive inner function that you call with start values to create the LazyList.

Exercise: Fibonacci LazyList

Implement a function on the companion, that generates an infinite LazyList of Fibonacci numbers (the *n*-th Fibonacci number is the sum of the previous two, with 0 and 1 as starting values):

$$F_0 = 0$$
$$F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2}$$

Tip: use a recursive inner function that you call with start values to create the LazyList.

```scala
def fibs =
  def go(f0: Int, f1: Int): LazyList[Int] =
    cons(f0, go(f1, f0+f1))
  go(0, 1)
```

Exercise: `unfold`

Implement a more general lazy-list-building function `unfold`. It takes an initial state and a function, that produces the next state and value in the lazy list or returns an empty optional when the lazy list should end:

```
def unfold[A, S](z: S)(f: S => Option[(A, S)]): LazyList[A]
```

As this function does not need an existing `LazyList`, add it to the companion object.

Corecursive functions

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

- `unfold` is a corecursive function:
    - Recursive function consumes data, corecursive function produces data
    - Corecursive functions need not terminate while *productive* (can evaluate more of the result in finite time)

Unfold: Usage example

```
def alphabet = unfold('a')(s =>
    if s <= 'z' then Some((s, (s+1).toChar))
    else None
    )
```

We start with the character `'a'`.

The function checks, if the current character is at most `'z'`. If yes, we return a tuple of it and the next character (which unfold will use on the next call to the function), wrapped in `Some`, as we want to go on.

If we reach a letter after `'z'` (in ASCII), we return `None` and end the lazy list.

Exercise: Fibonacci lazy list with `unfold`

Implement the fibonacci lazy list from before using unfold (*hint: as it should be inifinite, we never need to return* `None`).

Exercise: Fibonacci lazy list with `unfold`

Implement the fibonacci lazy list from before using unfold (*hint: as it should be inifinite, we never need to return* `None`).

Verbose solution:

```
def fibsViaUnfold =
  unfold( (0,1) ) { p =>
    p match
      //only case, p is always a tuple
      case (f0, f1) =>
        val currentElement = f0 // always return first tuple element
        val nextState = (f1, f0 + f1) // calculate next fibonacci number
        val result = (currentElement, nextState)
        Some(result) // always Some, lazylist is infinite
  }
```

Exercise: Fibonacci lazy list with `unfold`

Note: in a place, where a function literal is expected and the function begins with a pattern match on the parameter list, we can skip the parameter list and match:

```
{ case (f0,f1) => ...}
```

is equivalent to

```
p => p match { case (f0,f1) => ...}
```

So `fibsViaUnfold` can be shortened to:

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

LazyList in the standard library

Notable differences between our implementation and the LazyList in Scala's standard library:

- different internal structure
- Additional syntax for creating LazyLists, similar to normal lists:

  ```
  elemA #:: elemB #:: tail
  ```

  Syntax with `LazyList.cons` and `empty` is still available.
- *No* lazy fold. But available via external libraries, e.g. `cats.Foldable`.

Summary

- Non-strictness helps with efficiency of functional code
- It can help modularity by separating description and time of evaluation of an expression
- We can use `LazyList` mostly like `List`, to avoid unnecessary intermediate structures.
- Allows us to describe infinite sequences while staying referentially transparent.