Strictness And Laziness

ANKIT BAHUGUNA (ANKIT.BAHUGUNA@TERADATA.COM)

{11.09.2014}

What we will cover today?

- ✓Strict and Non-Strict (Lazy)Functions.
- ✓How laziness can be used to improve the efficiency and modularity of functional programs?
  - Lazy List / Streams
  - Memoisation Streams and Avoiding Re-Computation
  - Helper Functions for Inspecting Streams.
- ✓Separating Program Description from Evaluation.
- ✓Infinite Streams and Corecursion
- ✓Conclusion

Let’s Start with a Program Trace for List

```
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)
```

This view makes it clear how the calls to **map** and **filter** each perform their own traversal of the input and allocate lists for the output.

The catch?

Wouldn’t it be nice if we could somehow fuse sequences of transformations like this into a single pass and avoid creating temporary data structures?

And….

It turns out that we can accomplish this kind of automatic loop fusion through the use of non-strictness (or, less formally, laziness).

First: Let's Cover some Basics.

- ✓**Non-strictness** (function) means that the function may choose *not* to evaluate one or more of its arguments.
- ✓A ***strict*** function always evaluates its arguments. (sort of norm!)
- ✓Most languages only support functions that expect their arguments fully evaluated.
- ✓Unless we tell it otherwise, any function definition in Scala will be strict.

Example : Strict Function

```
def square(x: Double): Double = x * x
```

If we invoke:

- square(41.0 + 1.0) = 42.0*42.0 [STRICT]
- square(sys.error("failure")) = Exception!

But, We are already familiar with Non-Strictness!

The function && takes two Boolean arguments, but only evaluates the second argument if the first is true:

```
scala> false && { println("!!"); true } // does not print anything

res0: Boolean = false
```

And || only evaluates its second argument if the first is false:

```
scala> true || { println("!!"); false } // doesn't print anything either

res1: Boolean = true
```

+Example: non-strictness

```
val result = if (input.isEmpty) sys.error("empty
                    input") else input
```

This ‘if’ function would be non-strict, since it won’t evaluate all of its arguments.

In Scala, we can write non-strict functions by accepting some of our arguments unevaluated.

The Scala, if2[A]: Enters “thunk”

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

if2(a < 22,
 () => println("a"),
 () => println("b")

)
```

The arguments we’d like to pass unevaluated have a **() =>** immediately before their type.

A value of type () => A is a function that **accepts zero arguments** and returns an A.

*The unevaluated form of an expression is called a **thunk***, and we can ***force*** the thunk to evaluate the expression and get a result. We do so by invoking the function, passing an empty argument list, as in onTrue() or onFalse().

Cleaner Syntax

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

scala> if2(false, sys.error("fail"), 3)

res2: Int = 3
```

With either syntax, an argument that's passed unevaluated to a function will be evaluated once for each place it's referenced in the body of the function.

Scala won't cache the result of evaluating an argument!

Why Caching? : Example

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

maybeTwice: (b: Boolean, i: => Int)Int

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

hi

hi

x: Int = 84
```

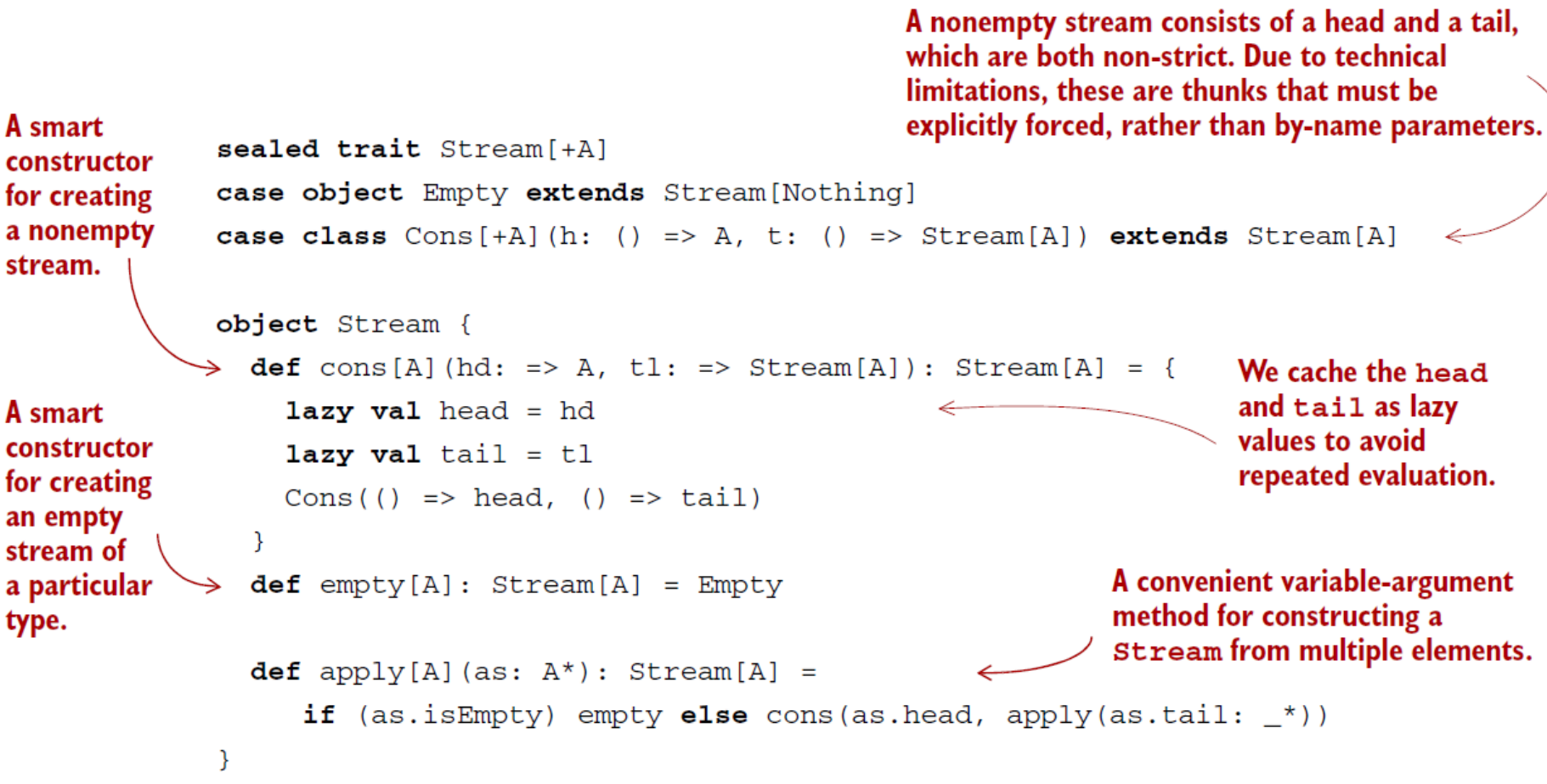
What's going on here?: **i** is referenced twice in the body of maybeTwice, and we've made it particularly obvious that it's evaluated each time by passing the block {println("hi"); 1+41}, which prints hi as a side effect before returning a result of 42. The expression 1+41 will be computed twice as well.

Caching the Value: Using **lazy** keyword!

```
scala> def maybeTwice2(b: Boolean, i: => Int) = {
| lazy val j = i
| if (b) j+j else 0
| }
maybeTwice: (b: Boolean, i: => Int)Int
scala> val x = maybeTwice2(true, { println("hi"); 1+41 })
hi
x: Int = 84
```

Adding the lazy keyword to a val declaration will cause Scala to delay evaluation of the right-hand side of that lazy val declaration until it's first referenced. It will also cache the result so that subsequent references to it don't trigger repeated evaluation.

Lazy Lists / Streams



EXAMPLE: Function to optionally extract the head of a Stream

```
def headOption: Option[A] = this match {
    case Empty => None
    case Cons(h, t) => Some(h())        //Explicit forcing of the 'h'
                                        //thunk using h()
}
```

Note: We have to force h explicitly via h() , but other than that, the code works the same way as it would for List . But this ability of Stream to evaluate only the portion actually demanded (We don’t evaluate the tail of the Cons) is useful!

Memoizing Streams and Avoiding Recomputation

We typically want to cache the values of a Cons node, once they are forced. If we use the Cons data constructor directly,
For instance, this code will actually compute expensive(x) twice:

```
val x = Cons(() => expensive(x), tl)

val h1 = x.headOption

val h2 = x.headOption
```

How to Avoid this? We define ***smart* constructors** i.e. function for constructing a data type that ensures some additional invariant or provides a slightly different signature than the "real" constructors used for pattern matching.

Convention, smart constructors typically *lowercase* the *first letter* of the corresponding data constructor in our case: **cons**

Smart Constructor: cons

Our **cons** smart constructor takes care of memoizing the by-name arguments for the head and tail of the Cons. This is a common trick, and it ensures that our thunk will only do its work once, when forced for the first time. Subsequent forces will return the cached lazy val:

```
def cons[A](hd: => A, tl: => Stream[A]): Stream[A] = {
    lazy val head = hd
    lazy val tail = tl
    Cons(() => head, () => tail)
}
```

Smart Constructor: empty

The **empty** smart constructor just returns **Empty**, but annotates Empty as a Stream[A], which is better for type inference in some cases. We can see how both smart constructors are used in the **Stream.apply** function:

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

Again, Scala takes care of wrapping the arguments to cons in thunks, so the as.head and apply(as.tail: _*) expressions won't be evaluated until we force the Stream.

Helper Function for Inspecting Streams

- ✓ **def** toList: List[A]: Convert Stream to List.
- ✓ take(n) : Returning the first n elements of a Stream.
- ✓ drop(n) : Skipping the first n elements of a Stream.
- ✓ **def** takeWhile(p: A => Boolean): Stream[A] : Returning all starting elements of a Stream that match the given predicate.

Separating Program Description from Evaluation

Major theme in functional programming:

"Separation of Concerns"

We want to separate the description of computations from actually running them.

Example(s)

1. First-class functions capture some computation in their bodies but only execute it once they receive their arguments.
2. We used Option to capture the fact that an error occurred, where the decision of what to do about it became a separate concern.
3. With Stream, we're able to build up a computation that produces a sequence of elements without running the steps of that computation until we actually need those elements.

In General*: Laziness let's us separate the description of an expression from the evaluation of that expression*. This gives us a powerful ability — we may choose to describe a "larger" expression than we need, and then ***evaluate only a portion of it***.

A Working Example

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

- ✓ p(h()) returns true, then exists terminates the traversal early and returns true as well.
- ✓ The tail of the stream is a lazy val. So not only does the traversal terminate early, the tail of the stream is never evaluated at all! So whatever code would have generated the tail is never actually executed.
- ✓ exists: Uses explicit recursion.

exists: Using General Recursion

If `f` doesn't evaluate its second argument, the recursion never occurs.

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

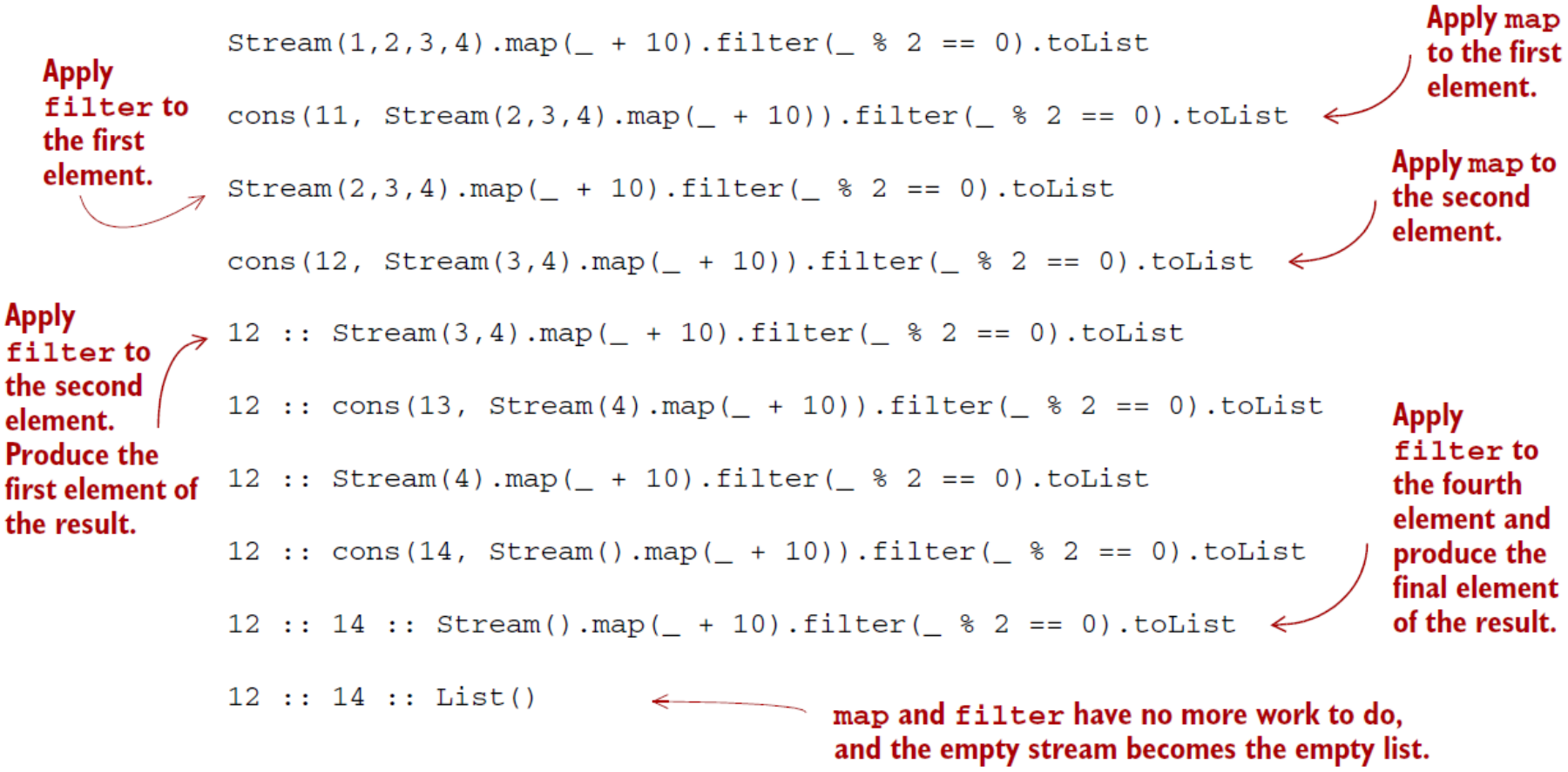
The arrow => in front of the argument type B means that the function `f` takes its second argument by name and may choose not to evaluate it.

foldRight to implement exists:

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

Here b is the unevaluated recursive step that folds the tail of the stream. If p(a) returns true, b will never be evaluated and the computation terminates early.

Program trace for **Stream: Re-Visited**



Explanation

Filter and map transformations are interleaved—the computation alternates between generating a single element of the output of map, and testing with filter to see if that element is divisible by 2 (adding it to the output list if it is).

Note:
We don't fully instantiate the intermediate stream that results from the map.

It's exactly as if we had interleaved the logic using a *special purpose loop*.

For this reason, people sometimes describe streams as **"first-class loops"** whose logic can be combined using higher-order functions like map and filter.

Benefits: Memory Usage.

As Intermediate streams aren't generated, a transformation of the stream requires only enough working memory to store and transform the current element.

Example: `Stream(1,2,3,4).map(_ + 10).filter(_ % 2 == 0)`

The garbage collector can reclaim the space allocated for the values 11 and 13 emitted by map as soon as filter determines they aren't needed.

Thus, for larger numbers of elements, Being able to reclaim this memory as quickly as possible can cut down on the amount of memory required by your program as a whole.

Infinite Streams and Corecursion

Example: An Infinite Stream of 1's

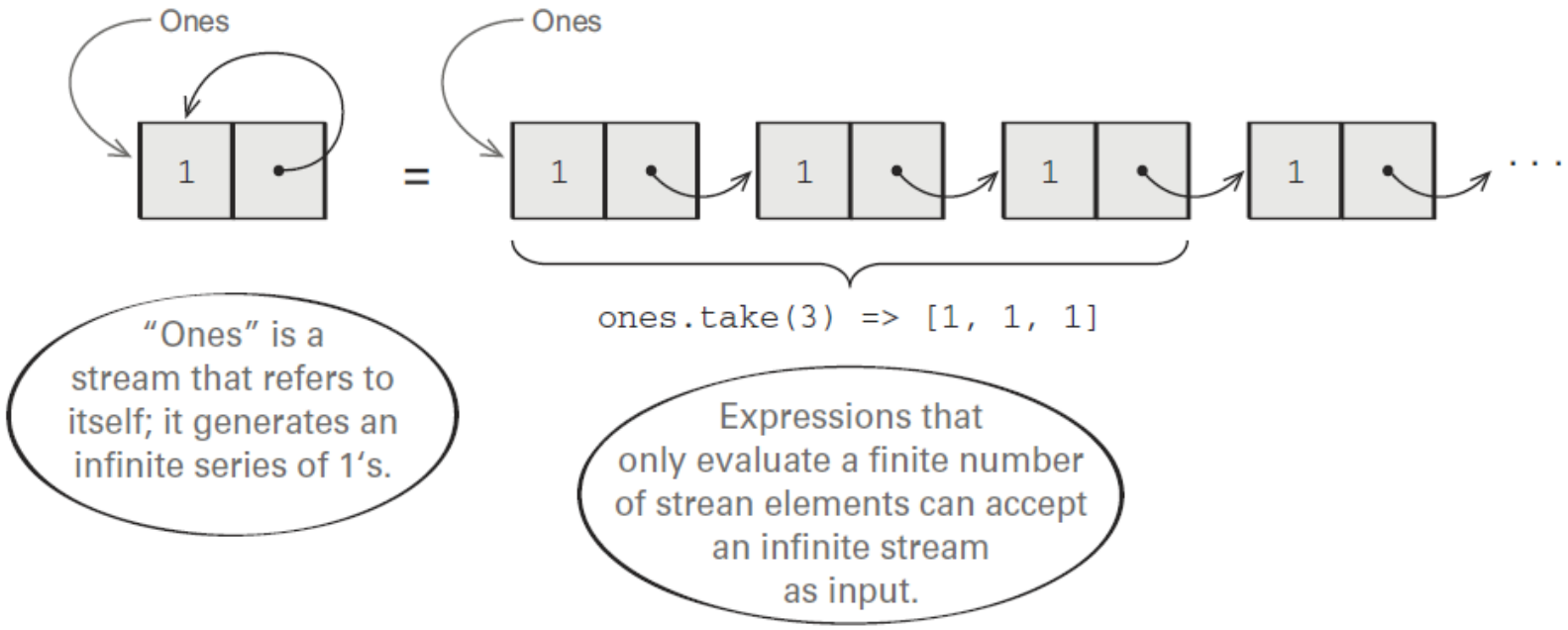
```
val ones: Stream[Int] = Stream.cons(1, ones)
```

But our functions work on infinite streams too.

```
scala> ones.take(5).toList
res0: List[Int] = List(1, 1, 1, 1, 1)
scala> ones.exists(_ % 2 != 0)
res1: Boolean = true
```

WARNING: Be Careful while using them with infinite streams since it's easy to write expressions that never terminate or aren't stack-safe.
Ex: **`ones.forAll(_ == 1)`**

An Infinite Stream of 1's



Many functions can be evaluated using finite resources even if their inputs generate infinite sequences.

unfold: A ‘more’ general stream building function

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

- ✓It takes an initial state, and a function for producing both the next state and the next value in the generated stream.
- ✓Option is used to indicate when the Stream should be terminated if at all.
- ✓The unfold function is an example of a ***corecursive*** function.
- ✓unfold is productive as long as f terminates, since we just need to run the function f one more time to generate the next element of the Stream

Recursive vs. Co-Recursive Function

Whereas a *recursive* function **consumes** data, a *corecursive* function ***produces*** data.

And whereas *recursive* functions **terminate** by recursing on smaller inputs, *corecursive* functions **need not terminate** so long as they remain *productive*, which just means that we can always evaluate more of the result in a finite amount of time.

- ✓ **Corecursion** sometimes called ***guarded recursion***, and
- ✓ **Productivity** sometimes called ***coterminination***.

Implement hasSubsequence using Laziness

```
def hasSubsequence[A](s: Stream[A]): Boolean =
     tails exists (_ startsWith s)
```

This implementation performs the **same number of steps** as a more monolithic implementation using nested loops with logic for breaking out of each loop early.

By using laziness, we can compose this function from simpler components and still retain the efficiency of the more specialized (and verbose) implementation.

Note: **tails** returns the Stream of suffixes of the input sequence, starting with the original Stream ; **startsWith** check if one Stream is a prefix of another.

Conclusion

- ✓Introduced non-strictness as a fundamental technique for implementing efficient and modular functional programs.
- ✓A bigger idea—non-strictness can **improve modularity** by separating the description of an expression from the how-and-when of its evaluation.
- ✓Keeping these concerns separate lets us **reuse a description in multiple contexts**, evaluating different portions of our expression to obtain different results.

NEXT CHAPTER: Purely functional approaches to *state*.

THANK YOU!