Streams and Laziness

Lecture 040 of Advanced Programming

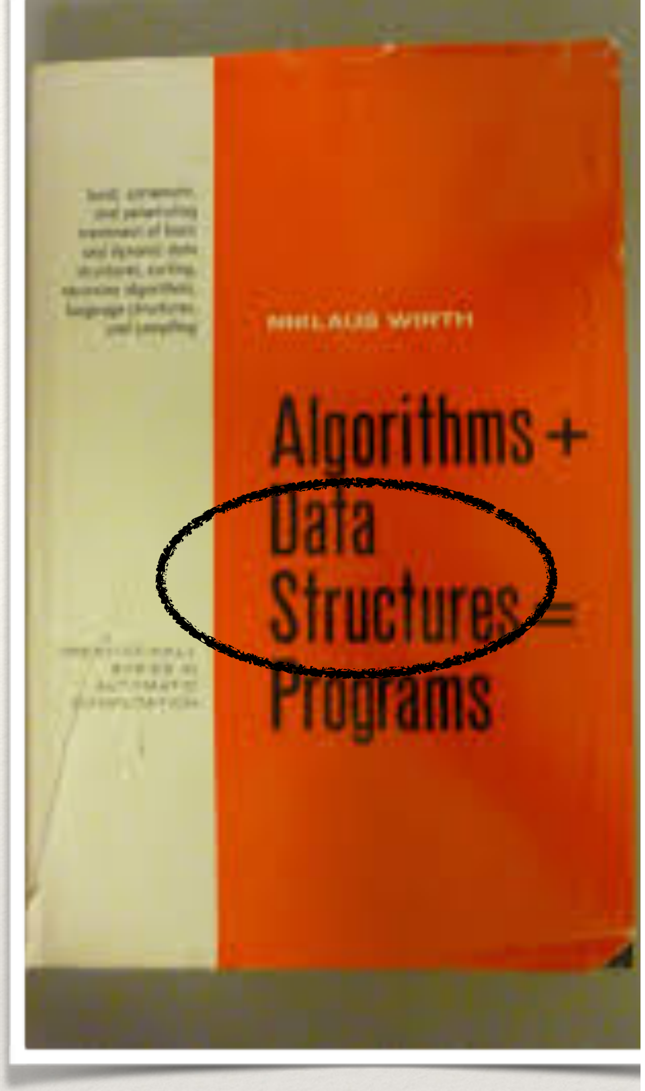
Feb. 21, 2018

Andrzej Wasowski & Zhoulai Fu

IT University of Copenhagen

Today's topic: Data Structures and Performance

- We have talked about functional Data structures such as List, Tree and operations such as filer, map
- Historically, they caused performance issues, making functional programming paradigms not so well accepted
- Advances in compiler technologies and data structures in FP have quickly changed that situation
- Today's topics:
  - Stream
  - Lazy Evaluation



Part 1: Laziness

Lazy: Disinclined to activity or Exertion.

e.g. The lazy child tried to avoid household chore

Dictionary of Merriam-Webster.

In this class: *The lazy list tries to avoid unnecessary computation.*

Motivation:
Data Structures and Performance

A Performance Issue

- Example: Find the 31-st prime number
- Use your laptop to find the answer, then we share the numbers we got, and time elapsed for the laptop to do the computing

DEMO

Your Takeaway

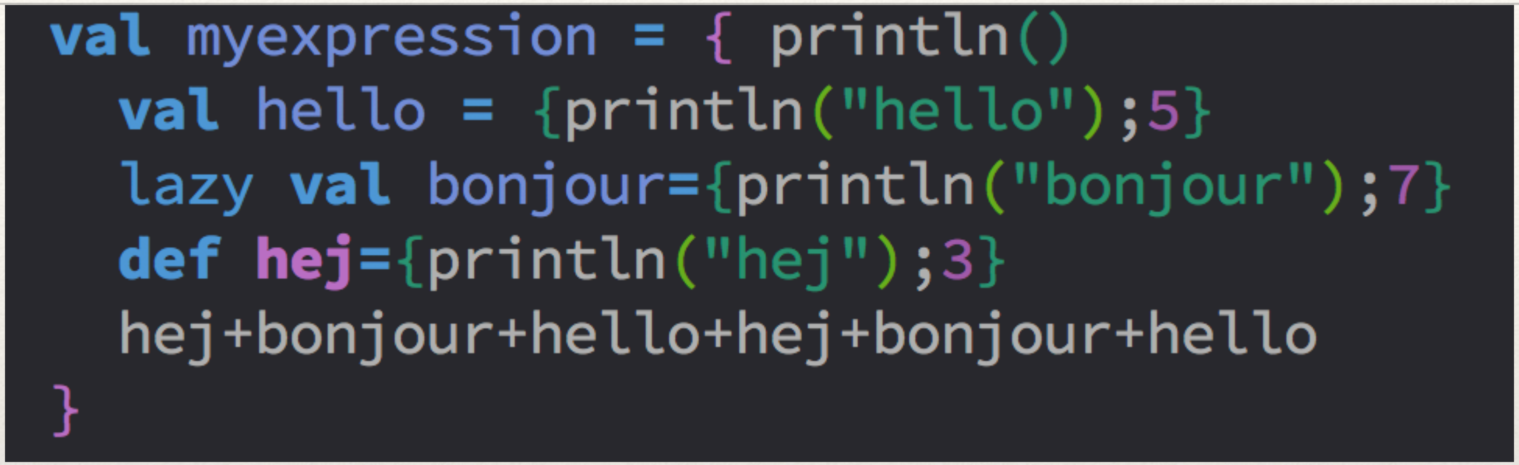
- Picking right (functional) data structures is essential for writing efficient (functional) programs
- Stream is a lazy version of list

Concepts

Three Kinds of Evaluation Strategies

- An evaluation strategy determines when an evaluation occurs
- By-value evaluation evaluates an expression to its value immediately, val x = {println(“hello”), 42}
- By-name evaluation evaluates an expression whenever it is accessed, e.g. def x = {println(“hello”), 42}
- By-need evaluation (or lazy evaluation) evaluates an expression when it is accessed the first time; the results will be cached afterwards, e.g. lazy val x = {println(“hello), 42}
- For all three, the expression “x” evaluates to 42 with side-effect “hello”, but they occur on different situations

Quiz



```
val myexpression = { println()
  val hello = {println("hello");5}
  lazy val bonjour={println("bonjour");7}
  def hej={println("hej");3}
  hej+bonjour+hello+hej+bonjour+hello
}
```

- What will be the output?
- Remind:
  - val: immediately
  - lazy val: first access
  - def: each access

DEMO

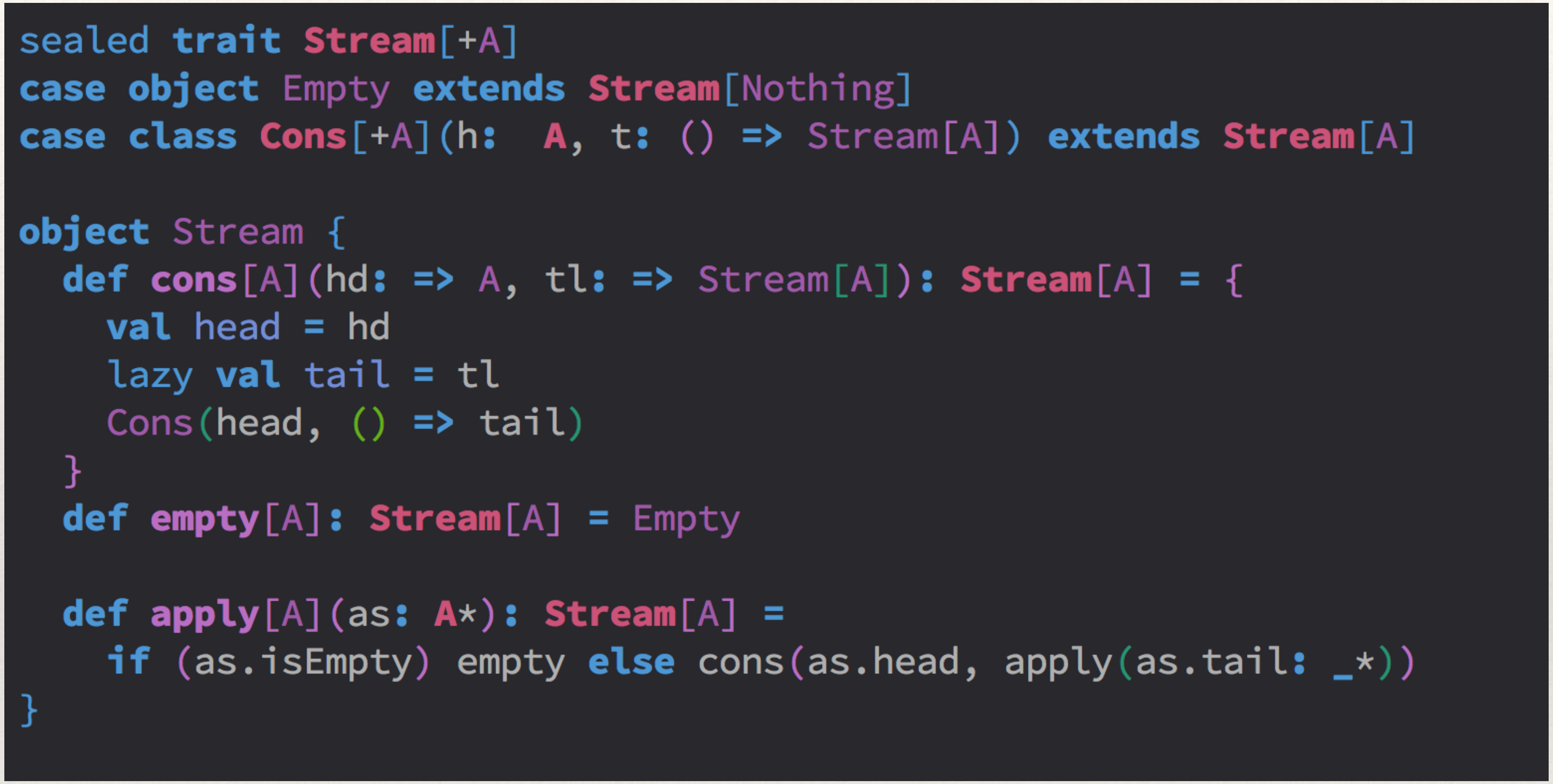
Strictness/Laziness

- We use the terms strictness/laziness on evaluation strategies of function calls
- A functions is strict if it evaluates all of its arguments
  - Scala functions are strict by default
- A function is non-strict (or lazy) if it may choose *not* to evaluate one or more of its arguments
  - &&, | |

DEMO

Implementation

We implement Stream as a List with a lazy tail



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

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

DEMO

Quiz

- Implement get[A](n:Int, s:Stream[A]): A that retrieves the nth item of stream s
- Implement filter[A](p: A => Boolean, s:Stream[A]): Stream[A]
- Implement streamRange[A](l:Int,h:Int):Stream[A] that gets the stream from l to h
- Test your implementation with this line: “get(30, filter (isPrime, streamRange(1,1000)))”

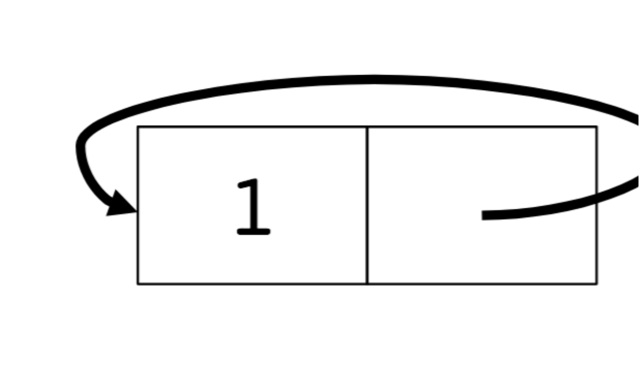
Performance comparison

- Compare the time of running
  - get(30, filter (isPrime, streamRange(1,1000))), and
  - (1 to 1000).filter(isPrime)(30)
- Think (again) why the former is faster

DEMO

Streams in the Real-World

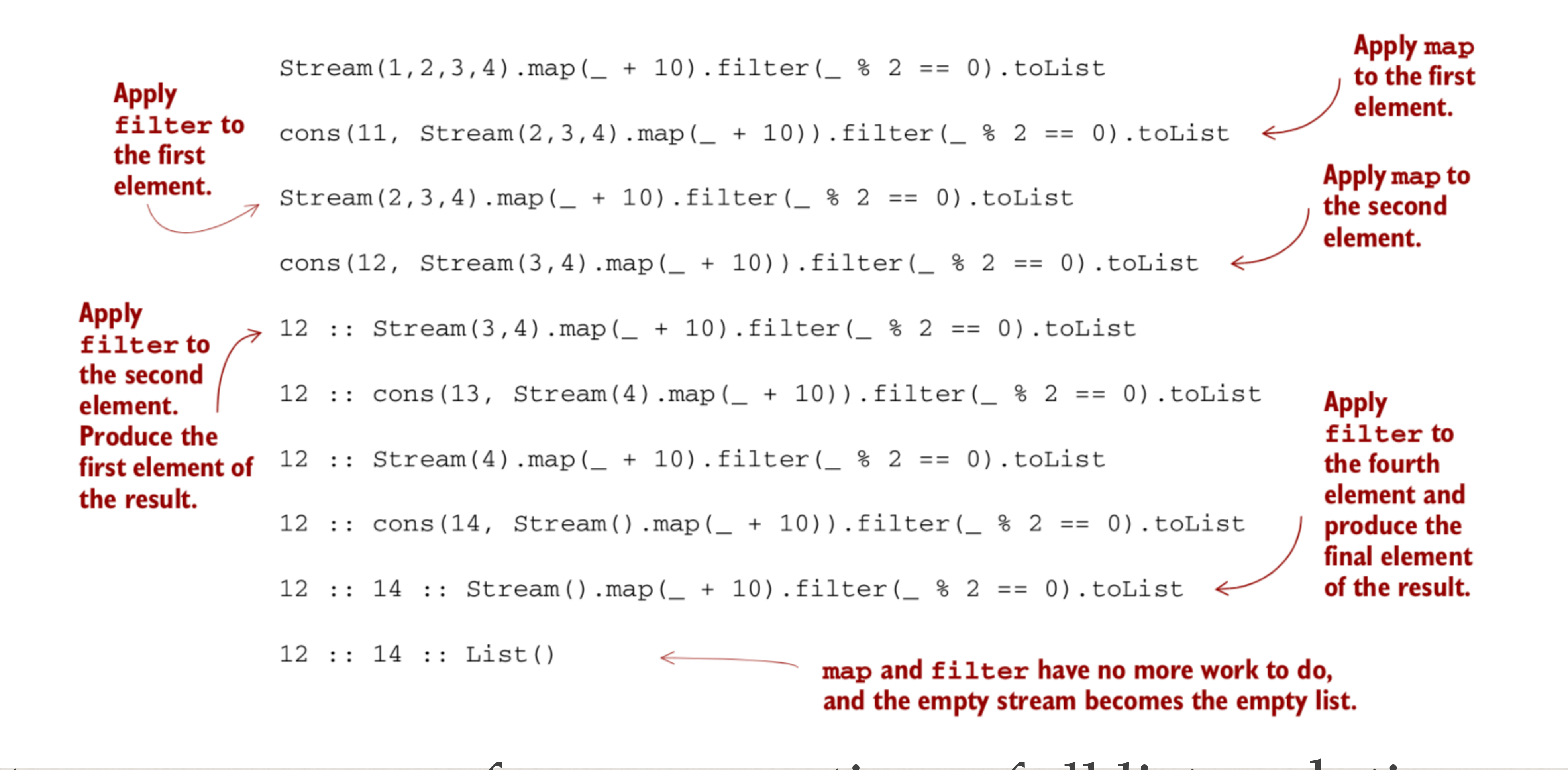
Infinite List



- val ones: Stream[Int] = Stream.cons(1, ones)
- ones(1000)
- Stream.from(1).filter(isPrime)(30) finds the 31st prime starting from 1, which avoids allocating an unnecessarily large list

Separating Program Description from Evaluation

- Laziness allows us to separate the description of an expression from the evaluation of the expression.
- Example: Stream(1,2,3,4).map(_ + 10).filter(_ % 2 == 0).toList



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

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

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

cons(12, Stream(3,4).map(_ + 10)).filter(_ % 2 == 0).toList

12 :: Stream(3,4).map(_ + 10).filter(_ % 2 == 0).toList

12 :: cons(13, Stream(4).map(_ + 10)).filter(_ % 2 == 0).toList

12 :: Stream(4).map(_ + 10).filter(_ % 2 == 0).toList

12 :: cons(14, Stream().map(_ + 10)).filter(_ % 2 == 0).toList

12 :: 14 :: Stream().map(_ + 10).filter(_ % 2 == 0).toList

12 :: 14 :: List()
```

Apply map to the first element.

Apply filter to the first element.

Apply map to the second element.

Apply filter to the second element. Produce the first element of the result.

Apply filter to the fourth element and produce the final element of the result.

map and filter have no more work to do, and the empty stream becomes the empty list.

Streams save you from generating a full list each time "map" or "filter" is invoked (see p72-73 of your textbook [Chiusano])

Conclusions

- Performance of Streams versus Lists
- By-value, by-name and by-need evaluations
- Strictness and non-strictness (laziness)
- Infinite Lists
- Separating program description from evaluation with laziness