@AndrzejWasowski
Andrzej Wąsowski

Advanced Programming

Laziness and Streams

IT UNIVERSITY OF COPENHAGEN

SOFTWARE
QUALITY
RESEARCH

AGENDA

- The idea of **delaying computations**
- **Evaluation strategies**:
  - Call-by-value (strict), call-by-name (non-strict), lazy (also non-strict)
- **Applications** of Call-By-Name and Laziness
- **Reactive Extensions**: A Motivating application of Streams (push)
- An implementation of **lazy streams** (pull)

I use imperative code (print) to demonstrate differences between call-by-value and call-by-name

We still disallow imperative code in home works, unless asked for it explicitly

Strict vs Non-strict Evaluation

- A function is **strict** if it always evaluates all its arguments (typically before evaluating its body)
- A **non-strict** function may choose not to evaluate all of its arguments.

Definition (strictness)

A function $f(x)$ is **strict** iff for every expression x that diverges (does not terminate or fails) the execution of $f(x)$ diverges, too.

- Strictness is a common default in most languages
- Haskell is non-strict, Scala is strict by default
- **Every language has a non-strict construct**
- Typical non-strict constructs: control flow statements, say **if-then-else**, and some operators disjunction, and conjunction
- **Every language needs a strict construct** (otherwise nothing will be computed)
- For example, pattern matching is strict in Haskell (and so it is in Scala).

Any functional language can simulate non-strictness

Lambda abstraction as a delay operator

- In any functional language non-strictness can be **simulated** quite easily
- Use `() => A` , a type of a nullary function returning `A`

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

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

- **Mentimeter:** What is the value of `res` if `a == 42` ? (think first, a trick)
- A **delayed computation** is called a **thunk**, executing a thunk is called **forcing** it
- Scala has special syntax to make it slightly nicer (**call-by-name**):

```
if2 (a < 22, println ("a"), println ("b"))
val y = if2 (a<22,f(x), g(x))
```

- The semantics of both programs are the same, but forcing is automatic, **no caching**

Lazy Evaluation

- A by-name argument of a function is re-evaluated **every single time it is accessed**
- Store it in a `lazy val` if you want to evaluate **only once**
- A lazy val is forced at first access, the value cached, retrieved on later accesses

```
def convoluted (a: => Int, b: => Int): Unit = {
  lazy val cacheB = b
  lazy val cacheA = a
  cacheA;
  cacheB;
  cacheA;
  cacheB;
}
convoluted (print ("A"), print ("B"))
```

- Prints `"AB"`
- Laziness interacts badly with **side effects**. Use in pure computations

Evaluation Strategies (Defs)

Definition (Call-by-value Evaluation)

The arguments of a function are evaluated before the function call. Then their **value** is substituted for the formal arguments

Definition (Call-by-name Evaluation)

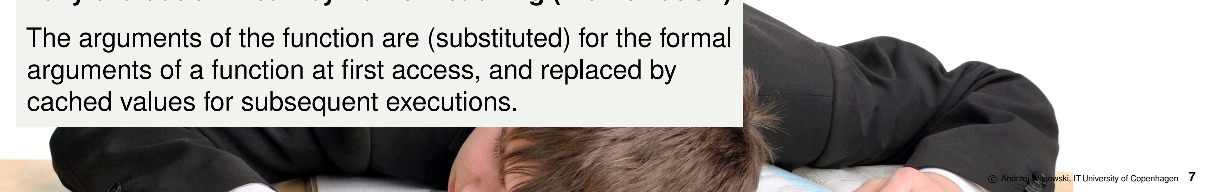
The arguments of a function are not evaluated but **syntactically substituted** for the formal arguments in the body

Definition (Lazy Evaluation)

Lazy evaluation = call-by-name + caching (memoization)

The arguments of the function are (substituted) for the formal arguments of a function at first access, and replaced by cached values for subsequent executions.

- Scala supports **all three** strategies
- In pure programs: **no difference** between these strategies (besides performance and memory usage)
- Impure programs: **perplexing** differences
- This difference allows the compiler and us to **simplify** and optimize pure programs
- For instance, constructing **only needed parts** of data structures



Call-by-name & Laziness: Usage

- Implementing **non-strict-functions**
  - If function **accesses parameters at most once** (simple control-flow like if, or, and, etc.) it can be built with call-by-name only, without memoization; for instance error handling code can be passed as one of parameters
  - We have seen: `getOrElse`, we can implement our own loops, etc.
- Implementing **internal DSLs** aka **fluent interfaces**
  - Call-by-name allows to hide the lambda expressions, when building **control-flow** constructs in an internal DSL [more next semester]
- Handling **large amounts of data**, only accessing necessary parts; especially when it is hard to see which parts need to be accessed/loaded/precomputed
- Implementing **generators of object/value sequences** elegantly (stream of naturals, stream of prime numbers, stream of messages, stream of random numbers, etc.)

Example of Call-by-name [1/4]

In Apache Spark's implementation

```
/**
 * Return a new RDD by applying a function to all elements of this RDD.
 */
def map[U: ClassTag](f: T => U): RDD[U] = withScope {
  val cleanF = sc.clean(f)
  new MapPartitionsRDD[U, T](this, (context, pid, iter) => iter.map(cleanF))
}
```

How do we implement a function as withScope?

With **call-by-value** the body of the block would always be executed.

But we can use **call-by-name**.

https://github.com/apache/spark/blob/master/core/src/main/scala/org/apache/spark/rdd/RDD.scala, captured 2017-02-25

Example of Call-by-name [2/4]

In Apache Spark's implementation

```scala
/**
 * Execute a block of code in a scope such that all new RDDs created in this body will
 * be part of the same scope. For more detail, see {{org.apache.spark.rdd.RDDOperationScope}}.
 *
 * Note: Return statements are NOT allowed in the given body.
 */
private[spark] def withScope[U](body: => U): U = RDDOperationScope.withScope[U](sc)(body)
```

The body (U) is passed by-name and then

Forwarded to a similar method in another class, also by-name

No forcing happens

`https://github.com/apache/spark/blob/master/core/src/main/scala/org/apache/spark/rdd/RDD.scala`, captured 2017-02-25

Example of Call-by-name [3/4]

In RDDOperationScope ...

```
private[spark] def withScope[T](
    sc: SparkContext,
    name: String,
    allowNesting: Boolean,
    ignoreParent: Boolean)(body: => T): T = {
  // Save the old scope to restore it later
  val scopeKey = SparkContext.RDD_SCOPE_KEY
  val noOverrideKey = SparkContext.RDD_SCOPE_NO_OVERRIDE_KEY
  val oldScopeJson = sc.getLocalProperty(scopeKey)
  val oldScope = Option(oldScopeJson).map(RDDOperationScope.fromJson)
  val oldNoOverride = sc.getLocalProperty(noOverrideKey)
  try {
    if (ignoreParent) {
      // Ignore all parent settings and scopes and start afresh with our own root scope
      sc.setLocalProperty(scopeKey, new RDDOperationScope(name).toJson)
    } else if (sc.getLocalProperty(noOverrideKey) == null) {
      // Otherwise, set the scope only if the higher level caller allows us to do so
```

Example of Call-by-name [4/4]

```
} else if (sc.getLocalProperty(noOverrideKey) == null) {
  // Otherwise, set the scope only if the higher level caller allows us to do so
  sc.setLocalProperty(scopeKey, new RDDOperationScope(name, oldScope).toJson)
}
// Optionally disallow the child body to override our scope
if (!allowNesting) {
  sc.setLocalProperty(noOverrideKey, "true")
}
body
```

The body is executed if the control flow reaches the last line above
(in here: no exceptions thrown)
Otherwise the body will never be executed

https://github.com/apache/spark/blob/master/core/src/main/scala/org/apache/spark/rdd/RDDOperationScope.scala, 2017-02-25

Lazy Streams (Pull Streams)

- Reactive programming streams are **push-streams** (react when a new event arrives)
- We now develop **pull-streams** (ask for data when we needed)

```
sealed trait Stream [+A] {
  def headOption[A] = this match {
     case Empty => None
     case Cons(h,t) => Some (h())
  }
}
case object Empty extends Stream[Nothing]
case class Cons[+A] (h: () => A, t: () => Stream[A]) extends Stream[A]
```

- Lazy streams are **isomorphic to lists**, but evaluate the head and tail arguments **lazily**
- Case classes cannot take by-name args in Scala, so we use the trick from `if2`
- We provide a **convenience constructor** to work around this limitation:

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

Examples with Lazy Streams

- An infinite stream of ones:

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

- We can implement similar methods as on lists. What difference does laziness bring?
- Chop `n` elements from a prefix of a stream:

```
def drop (n: Int): Stream[A] = ???
```

- Take a finite prefix from a stream (exercises):

```
def take (n: Int): Stream[A] = ???
```

- Convert a finite stream to a list (exercises)

```
def toList: List[A] = ???
```

- Two streams of random numbers:

```
val random1: Stream[Double] = Cons(() => Math.random, () => random1)
val random2: Stream[Double] = cons (Math.random, random2)
random1.take (5).toList      // try several times
random2.take (5).toList      // try several times
```

- `random1` always gives a new list of values, `random2` always the same list. Why?
- We see a difference because `random1` is **not referentially transparent**

Why Streams?

- Separate the **description of computation** from running it
- **Run only what you need**
  - Describe a larger expression, **evaluate only a portion** of it
  - Easier to program an infinite stream of random numbers than precisely 1000
  - The generator (stream) can be separated from the context of use: sometimes need 42 sometimes 1000 numbers, **generate them the same way**
  - Work with data **incrementally** as if everything loaded into memory
- **Fusion**: cache/memory locality
  - "`list map f map g`" runs two iterations over a list
  - "`stream map f map g`" runs one iteration over a stream
- **Streams are functional iterators.** In OO an iterator is an example of laziness. Forcing is explicit, usually called `next`
- Conceptual basis for **reactive programming** (+real time, push, etc.)