INFO0054-1
Programmation Fonctionnelle

Chapter 05: Evaluation strategies

Christophe Debruyne
(c.debruyne@uliege.be)

References

- Chapter 5: Strictness and laziness.
  Paul Chiusano and Runar Bjarnason. Functional Programming in Scala, Manning Publications, 2015.

- Recommended to consult:
  - Evaluation strategies in Scala by Tudor Zgureanu. PDF also available on eCampus.

Overview

- Evaluation orders and strategies
- Strict evaluation: call by value, call by reference
- Non-strict evaluation: call by name, call by need
- Non-strict evaluation in Scala
- Non-strict functions in FP
- Examples: lazy lists, and infinite streams

Part 1

Part 1:

Non-strict evaluation &

Non-strict evaluation in Scala

Evaluation orders and strategies I

- There are many evaluation strategies (see Wikipedia for an overview).

- In short, there are two evaluation orders:
  - Strict evaluation, also known as applicative order, means that arguments are evaluated before the function is applied.
  - In non-strict evaluation, also known as normal order, the evaluation of the arguments of a function call are delayed until they are needed.

- Some languages only (*) use strict evaluation (e.g., Java).
  - (*) In Java, one cannot define the evaluation strategy of functions. But like most programming languages, some of the operators are evaluated in a lazy manner. Examples are the Boolean operators && and ||
    - && will not evaluate its right operand if the left operand is `false`
    - || will not evaluate its right operand if the left operand is `true`
- Some languages only use non-strict evaluation (e.g., Haskell).
- We have seen in the previous chapter that Scala supports both.

Evaluation orders and strategies II

- Each evaluation order may be supported by multiple evaluation strategies:
  - Strict evaluation:
    - Call by value
    - Call by reference
    - …
  - Non-strict evaluation:
    - Call by name
    - Call by need
    - …

- You have seen some of these strategies in previous classes, especially call by value and call by reference. In this lesson, you will learn about call by name and call by need.
- We will exemplify these notions not only with some simple examples, but also by developing *lazy lists* and *infinite streams*.

Strict evaluation in Scala

- Scala supports call by value, which is also the default strategy.

- In call by value, the expressions that are passed as arguments are evaluated and their results are copied into the formal parameters of the function.

- The value that is copied is either a primitive (e.g., `Int`) or a pointer to an object (e.g., `Employee`).

By the way:

- *Like Scala, Java only supports call by value.*
- *Java allows you to overwrite the value of a parameter (and only the copy is overwritten).*
- *In Scala, however, parameters are `val` and cannot be changed.*
- *Overwriting the value of parameters in a call by value setting is considered a bad practice and Scala has decided to prohibit it by design. ;-)*

Call by value in Scala

```
def test(x: Int): Int =
    println("One")
    val y = x + x
    println("Two")
    y

----------

scala> test({ println("Test") ; 1 })
Test
One
Two
val res0: Int = 2
```

- We will use `println` in order to demonstrate the differences between evaluation strategies. Because of this, the code in these examples are not (that) pure and certainly not exemplary!
- `{ println("Test") ; 1}` is an expression (a block) containing two statements. When evaluating that block, it will first print a statement to the terminal before returning 1.
  - Remember that statements are separated by newlines or semicolons.
- As expected, Scala evaluates the argument before passing the result to the function.

Non-strict evaluation in Scala

- Scala supports call by name and call by need.

- Using a special prefix `=>`, we can state that an argument is called by name. This means we can substitute each occurrence of that variable with the expression that is passed.
  - We have seen examples of this special prefix in the previous class.

- Using the special keyword `lazy` before a `val` declaration, we can declare that the expression on the right-hand side will only be evaluated when used. I.e., call by need. The results are furthermore cached, a technique called memoization.

Call by name in Scala

```
def test(x: => Int): Int =
    println("One")
    val y = x + x
    println("Two")
    y

----------

scala> test({ println("Test") ; 1 })
One
Test
Test
Two
val res1: Int = 2
```

```
val y = x + x becomes
val y = { println("Test") ; 1} + { println("Test") ; 1}
```

- The arguments that we want to pass unevaluated have an => before their type.
- => "indicates that the corresponding argument is not evaluated at the point of function application, but instead is evaluated <u>at each use within the function</u>." (src)
  - In other words, you can replace all occurrences of x with the expression that is passed.

Call by need in Scala I

```
def test(x: Int): Int =
    println("One")
    lazy val y = { println("Foo"); x + x }
    println("Two")
    y

----------

scala> test({ println("Test") ; 1 })
Test
One
Two
Foo
val res2: Int = 2
```

- Lazy values are declared by placing the `lazy` keyword in front of a `val` declaration.
- The expression on the RHS of a `val` declaration will only be evaluated when the variable is used (i.e., needed).

Call by need in Scala II

```scala
def test(x: Int): Int =
    println("One")
    lazy val y = { println("Foo"); x + x }
    println("Two")
    val a = y + 1
    val b = y + 1
    a + b

----------

scala> test({ println("Test") ; 1 })
Test
One
Two
Foo
val res3: Int = 6
```

- Next to the lazy evaluation, Scala caches the results so that subsequent references do not lead to repeated evaluations.

Call by need in Scala III

```scala
def test(x: => Int): Int =
    println("One")
    lazy val y = x + x
    println("Two")
    y

----------

scala> test({ println("Test") ; 1 })
One
Two
Test
Test
val res4: Int = 2
```

By the way: call by need

- Call by need is not only used with call by name arguments. E.g.,

```
def div(n: Double, m: Double): Double =
    lazy val res = n / m
    if(m == 0.0) 0.0
    else res

----------

scala> div(10, 5)
val res0: Double = 2.0

scala> div(10, 0)
val res1: Double = 0.0
```

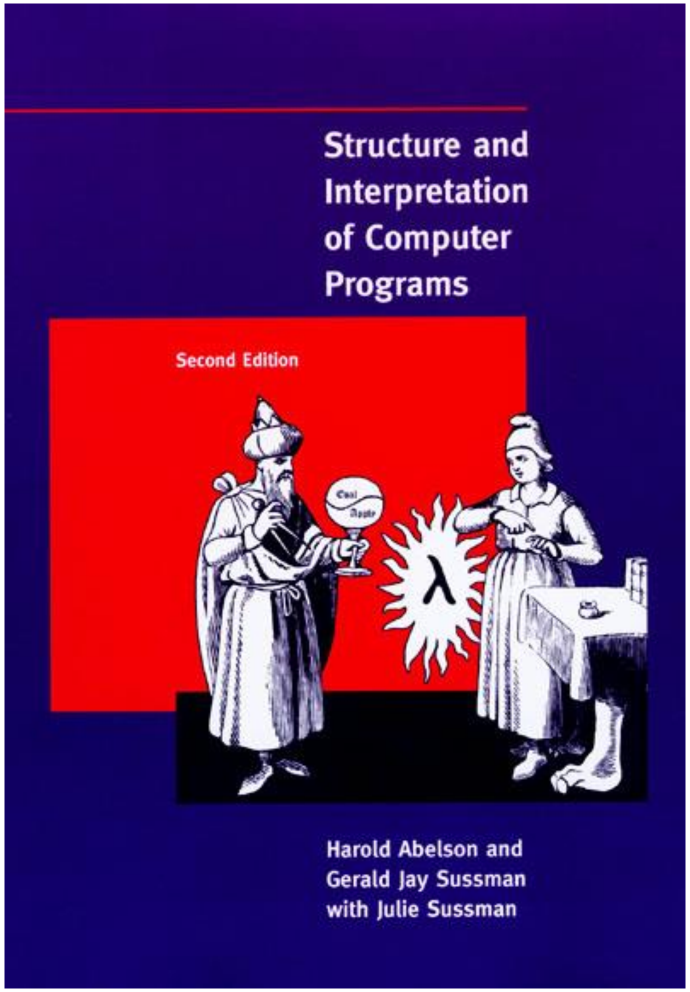
- **We know that sentinel values are to be avoided!** This example serves to illustrate how we can delay the evaluation of an expression. This may come in handy for storing intermediate results in variables but may impede the readability of your code!

By the way: a bit of terminology

Are non-strict evaluation and call by need the same? No. But, in practice you will find that:

"The difference between the “lazy” terminology and the “normal-order” terminology is somewhat fuzzy. Generally, “lazy” refers to the mechanisms of particular evaluators, while “normal-order” refers to the semantics of languages, independent of any particular evaluation strategy. But this is not a hard-and-fast distinction, and the two terminologies are often used interchangeably."

From: Abelson, Harold, and Gerald Jay Sussman. Structure and interpretation of computer programs. The MIT Press, 1996. https://web.mit.edu/6.001/6.037/sicp.pdf


Structure and Interpretation of Computer Programs
Second Edition
λ
Harold Abelson and Gerald Jay Sussman with Julie Sussman

Non-strict evaluation and FP

- Languages that support non-strict evaluation are usually functional programming languages. Some languages are non-strict by default (e.g., Haskell), other allows you to choose. Some other languages only allow it for certain data structures.

- The combination of non-strict evaluation with side effects (e.g., procedural coding and OOP) is difficult! Programmers should be very careful when combining the two. This is one of the reasons why non-strict evaluation is not that prevalent in imperative languages.

- When referring to call by name, we mentioned "replacing variables by their expressions." Indeed, non-strict evaluation is only correct and performant if referential transparency is respected.

Part 2

Part 2:

Non-strict evaluation in FP,
A motivating example

Non-strict functions in FP

Functions are either strict or non-strict. In other words, strictness and non-strictness are properties of functions. In the next few slides, we will see non-strict functions can drastically improve the performance of functional program.

Non-strictness is a fundamental technique for improving the efficiency and modularity of functional programs as it allows us to separate description of an expression from the evaluation of an expression.

A motivating example, chaining operations on lists:

```
scala> List(1, 2, 3, 4).map(math.pow(_,2)).filter(_ % 2 == 0).map(math.sqrt)
val res22: List[Double] = List(2.0, 4.0)
```

Each operation produces intermediate results which are used by a subsequent operation:

```
List(1, 2, 3, 4).map(math.pow(_,2)).filter(_ % 2 == 0).map(math.sqrt)
List(1.0, 4.0, 9.0, 16.0).filter(_ % 2 == 0).map(math.sqrt)
List(4.0, 16.0).map(math.sqrt)
List(2.0, 4.0)
```

Is there a way to "fuse" these operations so that only one pass is needed without rewriting the code? It turns out that non-strict evaluation allows us to do this.

Lazy lists

Lazy lists, also known as streams, are a data structure allowing us to fuse chains of operations into a single pass. In a stream, elements are evaluated only when they are needed.

Scala's standard library has support for streams, but we will implement our own stream as to gain a better understanding of non-strict evaluation. We will call our lazy list a Flux (the French term for stream in computing).

Flux I

- One can omit curly braces if they are carful with the code indentation.
- Notice the import statement. This allows us to use fields declared in our companion object without qualifying those field with `Flux`. I.e., rather than writing `Flux.empty`, we can write `empty`.
- We have created a variadic function with `apply` to create a constructor.

```
import Flux.*

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

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

  def toList: List[A] = this match
    case Cons(h,t) => h() :: t().toList
    case Empty => Nil

  def take(n: Int): Flux[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


object Flux:
  def cons[A](hd: => A, tl: => Flux[A]): Flux[A] =
    lazy val head = hd
    lazy val tail = tl
     Cons(() => head, () => tail)

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

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

Flux II

Scala prohibits the use of call by name for the constructor of data type constructors.

The reasons for those are quite technical, but long story short:

1. Scala does not guarantee immutability. The same expression may evaluate to different results.
2. If an expression is reused in several places, it may provide different results for each evaluation.
3. This is incompatible with data type constructors, which are meant for immutable data.

```scala
import Flux.*

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

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

  def toList: List[A] = this match
    case Cons(h,t) => h() :: t().toList
    case Empty => Nil

  def take(n: Int): Flux[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


object Flux:
  def cons[A](hd: => A, tl: => Flux[A]): Flux[A] =
    lazy val head = hd
    lazy val tail = tl
     Cons(() => head, () => tail)

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

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

Flux III

- We have two specializations of our Flux: Empty and Cons. But why do we need different "constructors" that are declared in the companion object?
- Scala does not allow call by name for case class constructors. This means with must use *thunks*. The disadvantage is that we may evaluate an expression multiple times
- The use of call by need in our smart constructors fixes that problem.

```scala
import Flux.*

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

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

  def toList: List[A] = this match
    case Cons(h,t) => h() :: t().toList
    case Empty => Nil

  def take(n: Int): Flux[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


object Flux:
  def cons[A](hd: => A, tl: => Flux[A]): Flux[A] =
    lazy val head = hd
    lazy val tail = tl
     Cons(() => head, () => tail)

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

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

Thunks are unevaluated forms of an expression that we use for delaying an evaluation.

head and tail are evaluated when they are requested. E.g., when we ask for a Cons's head.

Flux IV

```
scala> val x: Flux[Int] = Cons(() => { println("foo") ; 1 }, () => x)
val x: Flux[Int] = Cons(<<<OMITTED FOR BREVITY>>>)

scala> x.headOption
foo
val res0: Option[Int] = Some(1)

scala> x.headOption
foo
val res1: Option[Int] = Some(1)
```

Notice how the use of Cons, which returns us Cons objects, leads us to multiple evaluations of the same expression. When we use our smart constructors, however, the evaluation of these expressions are not only delayed, but also cached once the have been used.

```
scala> val y: Flux[Int] = cons({ println("foo") ; 1 }, y)
val y: Flux[Int] = Cons(<<<OMITTED FOR BREVITY>>>)

scala> y.headOption
foo
val res3: Option[Int] = Some(1)

scala> y.headOption
val res4: Option[Int] = Some(1)
```

Flux V

- Smart constructors thus allows us to avoid problem such as repeated evaluations.

- By convention, smart constructors have the same name as their data types but start with a lower case.
  - cons instead of Cons
  - empty instead of Empty
  - …

```
import Flux.*

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

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

  def toList: List[A] = this match
    case Cons(h,t) => h() :: t().toList
    case Empty => Nil

  def take(n: Int): Flux[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


object Flux:
  def cons[A](hd: => A, tl: => Flux[A]): Flux[A] =
    lazy val head = hd
    lazy val tail = tl
     Cons(() => head, () => tail)

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

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

Flux VI

The function toList uses Scala's List data structure. :: is the cons operator and Nil stands for the empty list.

Now let us implement some functions such as map and filter.

You will reimplement these functions as well as implement others in the exercise sessions.

```
import Flux.*

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

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

  def toList: List[A] = this match
    case Cons(h,t) => h() :: t().toList
    case Empty => Nil

  def take(n: Int): Flux[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


object Flux:
  def cons[A](hd: => A, tl: => Flux[A]): Flux[A] =
    lazy val head = hd
    lazy val tail = tl
     Cons(() => head, () => tail)

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

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

Flux: filter and map

```scala
    def filter(f: A => Boolean): Flux[A] = this match
         case Cons(h, t) if f(h()) => cons(h(), t().filter(f))
         case Cons(_, t) => t().filter(f)
         case _ => empty

    def map[B](f: A => B): Flux[B] = this match
        case Cons(h, t) => cons(f(h()), t().map(f))
        case _ => empty
```

```
----------

scala> Flux(1, 2, 3, 4).map(_ + 2).filter(_ % 2 == 0).toList
val res0: List[Int] = List(4, 6)
```

Flux: tracing filter and map I

```
scala> Flux(1, 2, 3, 4).map(_ + 2).filter(_ % 2 == 0).toList
val res0: List[Int] = List(4, 6)


Flux(1, 2, 3, 4).map(_ + 2).filter(_ % 2 == 0).toList         // start
cons(3, Flux(2, 3, 4).map(_ + 2)).filter(_ % 2 == 0).toList  // map on 1st element
Flux(2, 3, 4).map(_ + 2).filter(_ % 2 == 0).toList           // filter on 1st element
cons(4, Flux(3, 4).map(_ + 2)).filter(_ % 2 == 0).toList     // map on the 2nd element
cons(4, Flux(3, 4).map(_ + 2).filter(_ % 2 == 0)).toList     // filter on the 2nd element
4 :: Flux(3, 4).map(_ + 2).filter(_ % 2 == 0).toList         // toList on the 2nd element
4 :: cons(5, Flux(4).map(_ + 2)).filter(_ % 2 == 0).toList   // map on the 3rd element
4 :: Flux(4).map(_ + 2).filter(_ % 2 == 0).toList            // filter on the 3rd element
4 :: cons(6, Flux().map(_ + 2)).filter(_ % 2 == 0).toList    // map on the 4th element
4 :: cons(6, Flux().map(_ + 2).filter(_ % 2 == 0)).toList    // filter on the 4th element
4 :: 6 :: Flux().map(_ + 2).filter(_ % 2 == 0).toList         // toList on the 4th element
4 :: 6 :: empty.filter(_ % 2 == 0).toList                     // map on empty stream
4 :: 6 :: empty.toList                                        // filter on empty stream
4 :: 6 :: Nil                                                 // toList on empty stream
```

Flux: tracing filter and map II

```
scala> Flux(1, 2, 3, 4).map(_ + 2).filter(_ % 2 == 0).toList
val res0: List[Int] = List(4, 6)


Flux(1, 2, 3, 4).map(_ + 2).filter(_ % 2 == 0).toList
cons(3, Flux(2, 3, 4).map(_ + 2)).filter(_ % 2 == 0).toList
Flux(2, 3, 4).map(_ + 2).filter(_ % 2 == 0).toList
cons(4, Flux(3, 4).map(_ + 2)).filter(_ % 2 == 0).toList
cons(4, Flux(3, 4).map(_ + 2).filter(_ % 2 == 0)).toList
4 :: Flux(3, 4).map(_ + 2).filter(_ % 2 == 0).toList
4 :: cons(5, Flux(4).map(_ + 2)).filter(_ % 2 == 0).toList
4 :: Flux(4).map(_ + 2).filter(_ % 2 == 0).toList
4 :: cons(6, Flux().map(_ + 2)).filter(_ % 2 == 0).toList
4 :: cons(6, Flux().map(_ + 2).filter(_ % 2 == 0)).toList
4 :: 6 :: Flux().map(_ + 2).filter(_ % 2 == 0).toList
4 :: 6 :: empty.filter(_ % 2 == 0).toList
4 :: 6 :: empty.toList
4 :: 6 :: Nil
```

The `filter` and `map` transformations are interleaved just like we would have if we would have implemented the transformations using a while loop.

We do not instantiate the intermediate streams. This allows us to reuse functions without having to worry about unnecessary computations.

We reduce memory usage as we only need the memory to work with the current element. E.g., with "regular" lists, we must keep 5 and 6 in memory before filter is applied.

Infinite streams

- Thanks to the use of call by need, we can create infinite streams. Infinite streams are stream in which in element refers to a prior element, creating a loop.

- Let's put the following in the companion object

```
val ones: Flux[Int] = cons(1, ones)
val one234 : Flux[Int] = cons(1, cons(2, cons(3, cons(4, one234))))
```

- Some examples
  - `scala> ones.take(5).toList`
  - `val res0: List[Int] = List(1, 1, 1, 1, 1)`

  - `scala> one234.filter(_ % 2 == 1).map(math.pow(_,2)).take(10)`
  - `val res1: Flux[Double] = Cons(<<<OMITTED FOR BREVITY>>>)`

  - `scala> one234.filter(_ % 2 == 1).map(math.pow(_,2)).take(5).toList`
  - `val res2: List[Double] = List(1.0, 9.0, 1.0, 9.0, 1.0)`

Lexicon

- Cache – mémoire cache
- Call by name – appel par nom
- Call by reference – appel par référence
- Call by need – appel par nécessité
- Call by value – appel par valeur
- Evaluation order – ordre d'évaluation
- Evaluation strategy – stratégie d'évaluation
- Lazy evaluation – évaluation paresseuse
- Memoization – mémoïsation
- Stream – flux