Répondre à tous les quiz sur https://etc.ch/kcpC

Programmation Appliquée en Scala

Chapitre 8: LazyList et Evaluation Paresseuse

Prof. Nastaran Fatemi
Christopher Meier

Printemps 2025

Dans ce cours

- Collections Scala et opérations de recherche combinatoire
- Problème de performance
- Evaluation retardée
- Définition de LazyList
- Méthodes sur LazyList
- Implémentation de LazyList
- Evaluation paresseuse
- Lazy val et LazyList
- Listes infinis

Les collections Scala et la recherche combinatoire (1)

Dans les cours précédents nous avons étudié plusieurs collections immuables qui fournissent des opérations puissantes, comme les fonctions d'ordre supérieur et les expressions `for`.

Ces fonctions sont particulièrement utiles pour implémenter des algorithmes de **recherche combinatoire**: algorithmes de recherche généralement considérés difficiles, mais réalisés en explorant efficacement une espace de solutions large.

- Les 8 dames
- Tic tac toe
- Puissance 4
- Etc.

Les collections Scala et la recherche combinatoire (2)

Par exemple, pour trouver le deuxième nombre premier entre 1000 et 10’000, on peut écrire:

```
((1000 to 10000) filter isPrime)(1)
```

Ce qui est beaucoup plus court que son alternative récursive:

```
def secondPrime(from: Int, to: Int) = nthPrime(from, to, 2)
def nthPrime(from: Int, to: Int, n: Int): Int =
  if from >= to then throw Error(”no prime”)
  else if isPrime(from) then
    if n == 1 then from else nthPrime(from + 1, to, n - 1)
  else nthPrime(from + 1, to, n)
```

Problème de performance

Mais d'un point de vu de performance, l'expression

```
((1000 to 10000).filter(isPrime)(1)
```

a un problème grave: elle construit une liste de tous les nombres premiers entre 1000 et 10000 pour s'intéresser seulement aux deux premiers.

Si on diminue la borne supérieure, on pourrait accélérer la vitesse, mais au risque de complètement perdre le deuxième nombre premier.

Evaluation retardée

Cependant, on peut utiliser une astuce pour rendre efficace le code plus court:

En évitant de calculer les éléments d'une séquence jusqu'à ce qu'il soit nécessaire pour le résultat de l'évaluation (ce qui peut être jamais).

Cette idée est implémentée dans une nouvelle classe: `LazyList`.

Les LazyLists sont similaires aux listes, mais leurs éléments sont évalués seulement **à la demande**.

Définir les LazyLists

Les LazyLists sont définis à partir d'une constante `LazyList.empty` et le constructeur `LazyList.cons`.

Par exemple

```
val xs = LazyList.cons(1, LazyList.cons(2, LazyList.empty))
```

On peut aussi les construire comme les autres collections en utilisant l'objet `LazyList`:

```
LazyList(1, 2, 3)
```

On peut également appeler la méthode `to(LazyList)` sur une collection donnée pour la transformer en `LazyList`:

```
(1 to 1000).to(LazyList) //> res0: LazyList[Int] = LazyList(<not computed>)
```

LazyList Range

Prenons l'expression `(lo until hi).to(LazyList)`

On peut écrire une fonction pour génère ce LazyList directement (sans utiliser la méthode `to(LazyList)`):

```
def lazyRange(lo: Int, hi: Int): LazyList[Int] =
  if lo >= hi then LazyList.empty
  else LazyList.cons(lo, lazyRange(lo + 1, hi))
```

D'une manière similaire à une fonction qui génère une liste:

```
def listRange(lo: Int, hi: Int): List[Int] =
  if lo >= hi then Nil
  else lo :: listRange(lo + 1, hi)
```

Comparaison des deux fonctions

Ces deux dernières fonctions ont presque la même structure, mais elles s'évaluent totalement différemment:

- `listRange(start, end)` produit une liste avec $end - start$ éléments et la retourne
- `lazyRange(start, end)` retourne un seul objet de type `LazyList`. **Les éléments sont calculés que quand cela est nécessaire, à savoir quand on appelle la tête ou le tail du `LazyList`.**

Méthodes sur LazyLists

`LazyList` supporte presque toutes les méthodes de `List`.

Par exemple, pour trouver le deuxième nombre premier entre 1000 et 10000:

```
LazyList.range(1000,10000).filter(isPrime)(1)
```

L’opérateur Cons de LazyList

La seule exception majeur est `::`.

`x :: xs` génère toujours une liste et non un LazyList.

Il y a un opérateur alternatif pour générer un LazyList et celui-ci est `#::`

```
x #:: xs == LazyList.cons(x, xs)
```

On peut utiliser `#::` aussi bien dans les expressions que dans les motifs.

Implémentation des LazyLists (1)

L'implémentation des LazyLists est assez subtile.

Pour simplifier, on considère d'abord que les LazyLists ont seulement la queque parreseuse. `head` et `isEmpty` sont calculé quand LazyList est créee. Ceci n'est pas le vrai comportement de LazyList, mais rend l'implémentation plus facile à comprendre.

Voici le trait `LazyList`:

```
trait LazyList[+T] extends Seq[T]:
  def isEmpty: Boolean
  def head: T
  def tail: LazyList[T]
    ...
```

Comme pour les listes, toutes les autres méthodes peuvent être définies à partir de ces trois méthodes.

Implémentation des LazyLists (2)

Les implémentations concrètes de LazyLists sont définies dans l'objet compagnon de LazyList. Voici un premier extrait:

```
object LazyList:
  def cons[T](hd: T, tl: => LazyList[T]) = new LazyList[T]
    def isEmpty = false
    def head = hd
    def tail = tl
    override def toString = "LazyList(" + hd + ",? )"

  val empty = new LazyList[Nothing]
    def isEmpty = true
    def head = throw new NoSuchElementException("empty.head")
    def tail = throw new NoSuchElementException("empty.tail")
    override def toString = "LazyList()"
```

Différence de `List`

La seule différence importante entre l'implémentation de `List` et de `LazyList` (simplifiée) concerne `tl` qui est le deuxième paramètre de `LazyList.cons`.

Dans l'implémentation de `LazyList` il s'agit d'un passage de paramètre par nom (*call by name*).

C'est pour cela que le deuxième argument de `LazyList.cons` n'est pas évalué sur le point de l'appel.

Il est par contre évalué à chaque fois que quelqu'un appelle `tail` sur un objet `LazyList`.

D’autres méthodes de `LazyList`

Les autres méthodes de `LazyList` sont implémentées de façon similaire à leurs méthodes homologues dans `List`.

Par exemple, voici l’implémentation de `filter`:

```
trait LazyList[+T] extends Seq[T]:
  ...
  def filter(p: T => Boolean): LazyList[T] =
    if isEmpty then this
    else if p(head) then cons(head, tail.filter(p))
    else tail.filter(p)
```

Evaluation paresseuse

L'implémentation proposée pour LazyList empêche le calcule non-nécessaire des éléments de tail, tant qu'ils ne sont pas requis. Mais, elle souffre **d'un autre problème potentiel de performance**: si `tail` est appelé plusieurs fois, le LazyList correspondant est recalculé à chaque fois.

On peut éviter ce problème en stockant le résultat de la première évaluation de tail et en réutilisant le résultat stocké au lieu de le recalculer.

– Cette optimisation fait du sens car dans un langage purement fonctionnel, une expression produit le même résultat à chaque fois qu'elle est évaluée.

On appelle ce schéma **évaluation paresseuse** (au contraire de l'**évaluation par nom**, où tout est recalculé, et de l'**évaluation stricte** pour les paramètres normaux et les définitions avec val.)

Evaluation paresseuse

Laziness means do things as late as possible and never do them twice!

A lazy dog

Evaluation paresseuse en Scala

Haskel est un langage de programmation fonctionnelle qui utilise l'évaluation paresseuse par défaut.

Scala utilise par défaut l'évaluation stricte, mais permet l'évaluation paresseuse des définitions de valeur en utilisant la forme `lazy val`:

```
lazy val x = expr
```

En préfixant la définition `val` avec le modificateur `lazy`, l'expression d'initialisation expr, sera **seulement évaluée la première fois que `val` est utilisée**.

- Ceci est semblable à la situation où `x` est définie comme une méthode sans paramètre en utilisant **`def`** `x = expr`
- La différence de `lazy val` et `def` c'est que `lazy val` n'est jamais évaluée plus qu'une fois.

*Répond à la **question 1** sur https://etc.ch/kcpC*

Quiz

Prenons le programme suivant:

```
def expr =
  val x = { print(”x”); 1 }
  lazy val y = { print(”y”); 2 }
  def z = { print(”z”); 3 }
  z+y+x+z+y+x

expr
```

Si on exécute ce programme, que sera affiché comme effet secondaire de l’évaluation de `expr`?

- **A** zyxzyx
- **B** xzyx
- **C** xzyz
- **D** zyzz
- **E** Autre chose

Lazy val et LazyList

Si on utilise `lazy val` pour `tail`, l'implémentation de `LazyList.cons` devient plus efficace:

```
def cons[T](hd: T, tl: => LazyList[T]) = new LazyList[T]:
  def head = hd
  lazy val tail = tl
  ...
```

Donc au lieu de **`def`** `tail = t1`, on utilise **`lazy val`** `tail = t1`.

Avec ce changement, comme avant, on évalue le `tail` la 1ère fois que c'est demandé, mais cette-fois-ci on réutilise l'évaluation de `tail` les prochaines fois qu'elle sera demandée. Alors, on empêche les calcules non-nécessaires.

Trace d'évaluation (1)

Pour se convaincre que l'implémentation des LazyLists évite véritablement les calculs non-nécessaires, on suit la trace d'exécution de l'expression suivante:

```
lazyRange(1000, 10000).filter(isPrime).apply(1)
```

→
```
(if 1000 >= 10000 then empty  // by extending lazyRange
else cons(1000, lazyRange(1000 + 1, 10000))
.filter(isPrime).apply(1)
```

→
```
cons(1000, lazyRange(1000 + 1, 10000))  // by evaluating if
.filter(isPrime).apply(1)
```

Trace d'évaluation (2)

On renomme `cons(1000, lazyRange(1000 + 1, 10000))` C1.

```
C1.filter(isPrime).apply(1)
```

→

```
(if C1.isEmpty then C1  // by expanding filter
else if isPrime(C1.head) then cons(C1.head,C1.tail.filter(isPrime))
else C1.tail.filter(isPrime))
    .apply(1)
```

→

```
(if isPrime(C1.head) then cons(C1.head, C1.tail.filter(isPrime))
 else C1.tail.filter(isPrime))  // by evaluating if
.apply(1)
```

→

```
(if isPrime(1000) then cons(C1.head, C1.tail.filter(isPrime))
 else C1.tail.filter(isPrime))  // by evaluating head
.apply(1)
```

Trace d'évaluation (3)

→→
```
(if false then cons(C1.head, C1.tail.filter(isPrime)) // eval. isPrime
 else C1.tail.filter(isPrime))
.apply(1)
```

→
```
C1.tail.filter(isPrime).apply(1)  // by evaluating if
```

→→
```
lazyRange(1001, 10000)  // by evaluating tail
.filter(isPrime).apply(1)
```

La séquence d'évaluation continue jusqu'à ce que on trouve le 1er nombre premier

→→
```
lazyRange(1009, 10000)
.filter(isPrime).apply(1)
```

→→
```
cons(1009, lazyRange(1009 + 1, 10000))  // by eval lazyRange
.filter(isPrime).apply(1)
```

Trace d'évaluation (4)

On renomme `cons(1009, lazyRange(1009 + 1, 10000))` en `C2`.

```
C2.filter(isPrime).apply(1)
```

$\twoheadrightarrow$

```
cons(1009, C2.tail.filter(isPrime)).apply(1)  //by eval. filter
```

$\rightarrow$

```
if 1 == 0 then cons(1009, C2.tail.filter(isPrime)).head // by eval. apply
else cons(1009, C2.tail.filter(isPrime)).tail.apply(0)
```

Supposons que `apply` est défini comme ceci dans `LazyList[T]`:

```
def apply(n: Int): T =
  if n == 0 then head
  else tail.apply(n-1)
```

Trace d'évaluation (5)

On renomme `cons(1009, lazyRange(1009 + 1, 10000))` en `C2`.

```
C2.filter(isPrime).apply(1)
```

$\twoheadrightarrow$

```
cons(1009, C2.tail.filter(isPrime)).apply(1)  //by eval. filter
```

$\rightarrow$

```
if 1 == 0 then cons(1009, C2.tail.filter(isPrime)).head // by eval. apply
else cons(1009, C2.tail.filter(isPrime)).tail.apply(0)
```

$\rightarrow$

```
C2.tail.filter(isPrime).apply(0)  // by eval. tail
```

$\rightarrow$

```
lazyRange(1010, 10000).filter(isPrime).apply(0)  // by eval. tail
```

Trace d'évaluation (6)

La séquence d'évaluation continue jusqu'à ce qu'on a le 2ème nombre premier:

$\rightarrow$
```
lazyRange(1013, 10000).filter(isPrime).apply(0)
```

$\rightarrow$
```
cons(1013, lazyRange(1013 + 1, 10000))  // by eval. lazyRange
.filter(isPrime).apply(0)
```

On renomme `cons(1013, lazyRange(1013 + 1, 10000)` en `C3`

$=$
```
C3.filter(isPrime).apply(0)
```

$\twoheadrightarrow$
```
cons(1013, C3.tail.filter(isPrime)).apply(0)  // by eval. Filter
```

$\rightarrow$
```
1013  // by eval. apply
```

On voit que seule la partie de LazyList qui est nécessaire pour le calcul du résultat a été construite.

Les listes infinis

On a vu que les éléments d'un LazyList sont calculés seulement quand ils sont requis pour produire un résultat.

On pourrait donc définir des listes infinis!

Par exemple, voici une liste de tous les entiers commençant par un nombre donné:

```
def from(n: Int ): LazyList [Int] = n #:: from(n+1)
```

La liste de tous les nombres naturels:

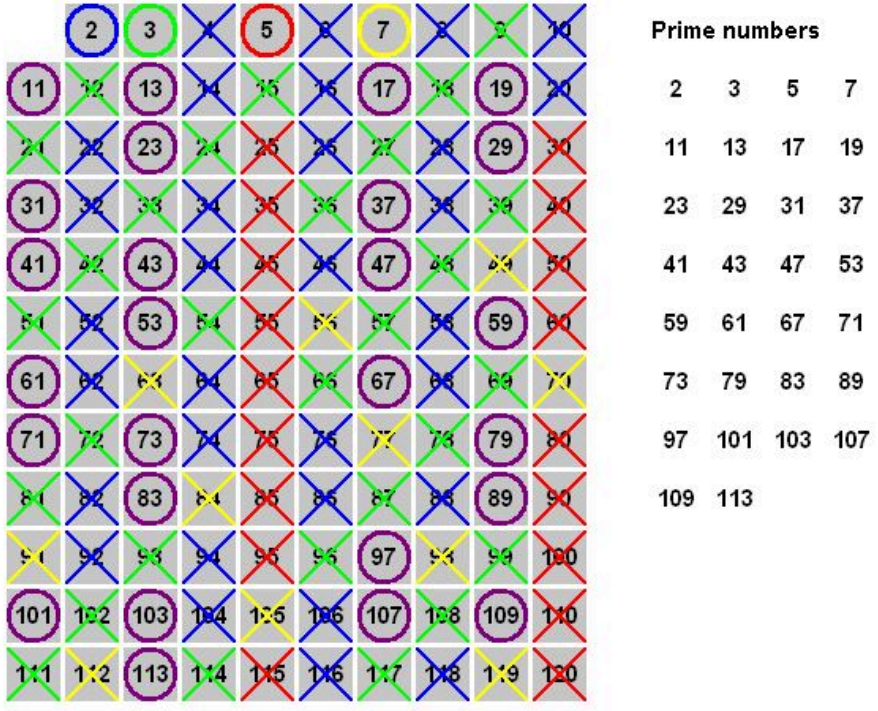
```
val nats = from(0)
```

La liste de tous les multiples de 4:

```
nats map (_ * 4)
```

Crible d’Ératosthène

Le crible d’Ératosthène est un procédé qui permet de trouver tous les nombres premiers inférieurs à un certain entier naturel donné N.



Crible d’Ératosthène

Crible d'Ératosthène

L'idée est la suivante:

- Commencer avec les nombres entiers, à partir de 2, le premier nombre premier.
- Eliminer tous les multiples de 2.
- Le premier élément de liste de résultat est 3, un nombre premier.
- Eliminer tous les multiples de 3.
- Itérer toujours. A chaque étape, le premier nombre dans la liste est un nombre premier
- Et on élimine tous ces multiples.

Crible d'Ératosthène: le code

Voici une fonction qui implémente ce principe:

```
def sieve(s: LazyList[Int]): LazyList[Int] =
  s.head #:: sieve(s.tail.filter(_ % s.head != 0))

val primes = sieve(from(2))
```

Pour voir une liste des N premiers nombres primaires, on peut écrire:

```
(primes.take(N)).toList
```

HEIG VD

*Répond à la **question 2** sur https://etc.ch/kcpC*

Quiz

Prenons les deux expressions suivantes qui expriment les listes infinis de multiples d'un nombre donné N:

```
val xs = from(1).map(_ * N)
val ys = from(1).filter(_ % N == 0)
```

Quel LazyList génère ces résultats plus rapidement?

A Le premier
B Le deuxième

Exercice

1. Ecrire une fonction qui prend un LazyList en paramètre et le transforme en une List:

```
def toList[A] (s:LazyList[A]): List[A] = ???
```

2. Ecrire une fonction qui prend un LazyList en paramètre et retourne le LazyList contenant seulement les n premiers éléments.

```
def take [A] (s: LazyList[A], n: Int) : LazyList[A] = ???
```