

Semaine 5 : Complément sur les listes

Réduction de listes

Une autre opération commune sur les listes est de combiner les éléments d'une liste avec un opérateur donné.

Par exemple :

$$\begin{aligned} \text{sum}(\text{List}(x_1, \dots, x_n)) &= 0 + x_1 + \dots + x_n \\ \text{product}(\text{List}(x_1, \dots, x_n)) &= 1 * x_1 * \dots * x_n \end{aligned}$$

On peut implanter cela en utilisant le schéma récursif habituel :

```
def sum(xs: List[Int]): Int = xs match {  
  case Nil  $\Rightarrow$  0  
  case y :: ys  $\Rightarrow$  y + sum(ys)  
}  
  
def product(xs: List[Int]): Int = xs match {  
  case Nil  $\Rightarrow$  1  
  case y :: ys  $\Rightarrow$  y * product(ys)  
}
```

La méthode générique `reduceLeft` insère un opérateur binaire donné entre deux éléments adjacents.

Par ex.

$$\text{List}(x_1, \dots, x_n).\text{reduceLeft}(op) = (\dots(x_1 \text{ op } x_2) \text{ op } \dots) \text{ op } x_n$$

Il est maintenant possible d'écrire plus simplement :

```
def sum(xs: List[Int]) = (0 :: xs) reduceLeft {(x:Int, y:Int) => x + y}
def product(xs: List[Int]) = (1 :: xs) reduceLeft {(x:Int, y:Int) => x * y}
```

Implantation de ReduceLeft

Comment peut-on implanter `reduceLeft` ?

```
abstract class List[a] { ...
  def reduceLeft(op: (a, a) => a): a = this match {
    case Nil => error("Nil.reduceLeft")
    case x :: xs => (xs foldLeft x)(op)
  }
  def foldLeft[b](z: b)(op: (b, a) => b): b = this match {
    case Nil => z
    case x :: xs => (xs foldLeft op(z, x))(op)
  }
}
```

La fonction `reduceLeft` est définie en termes d'une autre fonction souvent utile, `foldLeft`.

foldLeft prend comme paramètre additionnel un *accumulateur* *z*, qui est retourné pour les listes vides.

Autrement dit,

$$(List(x_1, \dots, x_n) \text{ foldLeft } z)(op) = (\dots(z \text{ op } x_1) \text{ op } \dots) \text{ op } x_n$$

sum et *product* peuvent alors aussi être définies comme suit.

$$\begin{aligned} \mathbf{def} \text{ sum}(xs: List[Int]) &= (xs \text{ foldLeft } 0) \{(x, y) \Rightarrow x + y\} \\ \mathbf{def} \text{ product}(xs: List[Int]) &= (xs \text{ foldLeft } 1) \{(x, y) \Rightarrow x * y\} \end{aligned}$$

FoldRight et ReduceRight

Les applications de *foldLeft* et *reduceLeft* se déplient en arbres qui penchent vers la gauche :

Elles ont deux fonctions duales, *foldRight* et *reduceRight*, qui produisent des arbres qui penchent vers la droite. C.-à-d. :

$$\begin{aligned} List(x_1, \dots, x_n).reduceRight(op) &= x_1 \text{ op } (\dots (x_{n-1} \text{ op } x_n)\dots) \\ (List(x_1, \dots, x_n) \text{ foldRight } acc)(op) &= x_1 \text{ op } (\dots (x_n \text{ op } acc)\dots) \end{aligned}$$

Elles sont définies ainsi

```

def reduceRight(op: (a, a) => a): a = this match
  case Nil => error("Nil.reduceRight")
  case x :: Nil => x
  case x :: xs => op(x, xs.reduceRight(op))
}
def foldRight[b](z: b)(op: (a, b) => b): b = this match {
  case Nil => z
  case x :: xs => op(x, (xs foldRight z)(op))
}

```

Pour les opérateurs *op* associatifs et commutatifs, *foldLeft* et *foldRight* sont équivalents (même s'il peut y avoir une différence d'efficacité).

Mais parfois, seul l'un des deux opérateurs est approprié ou a le bon type.

Exemple : Voici une autre formulation de *concat* :

```

def concat[a](xs: List[a], ys: List[a]): List[a] =
  (xs foldRight ys) {(x, xs) => x :: xs}

```

Ici il n'est pas possible de remplacer *foldRight* par *foldLeft* (pourquoi ?).

7

Retour sur le renversement de listes

Voici une fonction de renversement de liste avec un coût linéaire.

L'idée est d'utiliser l'opération *foldLeft* :

```

def reverse[a](xs: List[a]): List[a] = (xs foldLeft z?)(op?)

```

Il ne reste plus qu'à remplir les parties *z?* et *op?*.

Essayons de les déduire à partir d'exemples.

Tout d'abord,

Cas de base: *List()*

```

reverse(List())           (par spécification de reverse)
= (List() foldLeft z)(op) (par définition de reverse)
= z                       (par définition de foldLeft)

```

Par conséquent, $z = List()$.

8

Ensuite,

Etape d'induction: $List(x)$

```
reverse(List(x))                (par spécification de reverse)
= (List(x) foldLeft List())(op) (par déf. de reverse avec z = List())
= op(List(), x)                 (par définition de foldLeft)
```

Par conséquent, $op(List(), x) = List(x) = x :: List()$. Cela suggère de prendre pour op l'opérateur $::$ en échangeant ses opérands.

On arrive donc à l'implantation suivante de *reverse*.

```
def reverse[a](xs: List[a]): List[a] =
  (xs foldLeft List[a]()) { (xs, x) => x :: xs }
```

Remarque : le paramètre de type dans $List[a]()$ est nécessaire pour l'inférence de types.

Q : Quelle est la complexité de cette implantation de *reverse* ?

Complément sur Fold et Reduce

Exercice : Complétez les définitions suivantes, basées sur l'utilisation de *foldRight*, qui introduisent des opérations de base pour manipuler les listes.

```
def mapFun[a, b](xs: List[a], f: a => b): List[b] =
  (xs foldRight List[b]()) { ?? }
```

```
def lengthFun[a](xs: List[a]): Int =
  (xs foldRight 0) { ?? }
```

Traitements imbriqués sur les listes

On peut étendre l'utilisation des fonctions d'ordre supérieur sur les listes à de nombreux calculs qui sont habituellement exprimés à l'aide de boucles imbriquées.

Exemple : Étant donné un entier positif n , trouver tous les couples d'entiers positifs i et j , avec $1 \leq j < i < n$ tels que $i + j$ soit premier.

Par exemple, si $n = 7$, les couples recherchés sont

i	2	3	4	4	5	6	6
j	1	2	1	3	2	1	5
$i + j$	3	5	5	7	7	7	11

Une manière naturelle de faire cela consiste à :

- Générer la suite de tous les couples d'entiers (i, j) tels que $1 \leq j < i < n$.
- Filtrer les couples pour lesquels $i + j$ est premier.

Une manière naturelle de générer la suite des couples est de :

- Générer tous les entiers i compris entre 1 et n (exclu). Cela peut être réalisé par la fonction

```
def range(from: Int, end: Int): List[Int] =  
  if (from ≥ end) List()  
  else from :: range(from + 1, end);
```

qui est prédéfinie dans le module `List`.

- Pour chaque entier i , générer la liste des couples $(i, 1), \dots, (i, i-1)$. On peut y arriver en combinant `range` et `map` :

```
List.range(1, i) map (x ⇒ (i, x))
```

- Finalement, combiner toutes les sous-listes en utilisant `foldRight` avec `...`

En rassemblant les morceaux on obtient l'expression suivante :

```
List.range(1, n)
  .map(i => List.range(1, i).map(x => (i, x)))
  .foldRight(List[(Int, Int)]()) {(xs, ys) => xs ::: ys}
  .filter(pair => isPrime(pair._1 + pair._2))
```

La fonction *flatMap*

La combinaison consistant à appliquer une fonction aux éléments d'une liste puis à concaténer les résultats est si commune que l'on a introduit une méthode spéciale pour cela dans *List.scala* :

```
abstract class List[a] { ...
  def flatMap[b](f: a => List[b]): List[b] = this match {
    case Nil => Nil
    case x :: xs => f(x) ::: (xs flatMap f)
  }
}
```

Avec *flatMap*, on aurait pu écrire une expression plus concise :

```
List.range(1, n)
  .flatMap(i => List.range(1, i).map(x => (i, x)))
  .filter(pair => isPrime(pair._1 + pair._2))
```

Q : Trouvez une manière concise de définir *isPrime* ? (Indice : utilisez *forall* définie dans *List*).

La fonction *zip*

La méthode *zip* dans la classe *List* combine deux listes en une liste de couples.

```
abstract class List[a] { ...
  def zip[b](that: List[b]): List[(a,b)] =
    if (this.isEmpty || that.isEmpty) Nil
    else (this.head, that.head) :: (this.tail zip that.tail);
```

Exemple : En utilisant *zip* et *foldLeft*, on peut définir le produit scalaire de deux listes de la manière suivante.

```
def scalarProduct(xs: List[Double], ys: List[Double]): Double =
  (xs zip ys)
  .map(xy => xy._1 * xy._2)
  .foldLeft(0.0){(x, y) => x + y}
```

Résumé

- Nous avons vu que les listes étaient une structure de données fondamentale en programmation fonctionnelle.
- Les listes sont définies par des classes paramétrées et sont manipulées par des méthodes polymorphes.
- Les listes sont aux langages fonctionnels ce que sont les tableaux aux langages impératifs.
- Mais contrairement aux tableaux, on n'accède généralement pas aux éléments d'une liste par leur indice.
- On préfère traverser les listes récursivement ou via des combinateurs d'ordre supérieur tels que *map*, *filter*, *foldLeft* ou *foldRight*.

Raisonnement sur les listes

Rappelons nous l'opération de concaténation pour les listes (voir sem. 4):

```
class List[a] {  
  ...  
  def :: (that: List[a]): List[a] = that match {  
    case Nil => this  
    case x :: xs => x :: (xs :: this)  
  }  
}
```

On aimerait vérifier que la concaténation est associative, et qu'elle admet la liste vide $List()$ comme élément neutre à gauche et à droite :

$$\begin{aligned}(xs :: ys) :: zs &= xs :: (ys :: zs) \\ xs :: List() &= xs = List() :: xs\end{aligned}$$

Q : Comment peut-on prouver des propriétés comme celles-ci ?

R : Par induction structurelle sur les listes.

Rappel : Induction naturelle (ou récurrence)

Rappelons le principe des preuves par induction naturelle :

Pour montrer une propriété $P(n)$ pour tous les entiers $n \geq b$,

1. montrer qu'on a $P(b)$ (cas de base),
2. pour tout entier $n \geq b$ montrer que :
si l'on a $P(n)$, alors on a aussi $P(n + 1)$
(étape d'induction).

Exemple : Étant donné

```
def factorial(n: Int): Int =  
  if (n == 0) 1 /* 1ère clause */  
  else n * factorial(n-1) /* 2ième clause */
```

montrer que, pour tout $n \geq 4$,

$$factorial(n) \geq 2^n$$

Cas de base: 4

Ce cas est établi par simples calculs de $\text{factorial}(4) = 24$ et $2^4 = 16$.

Etape d'induction: $n+1$ On a pour $n \geq 4$:

$$\begin{aligned} & \text{factorial}(n + 1) \\ = & (n + 1) * \text{factorial}(n) && \text{(par la 2ième clause de factorial (*))} \\ \geq & 2 * \text{factorial}(n) && \text{(par calcul)} \\ \geq & 2 * 2^n. && \text{(par hypothèse d'induction)} \end{aligned}$$

Remarquez que dans une preuve on peut librement appliquer des étapes de réduction comme (*) à l'intérieur d'un terme.

Ça fonctionne parce que les programmes fonctionnels purs n'ont pas d'effets de bord ; si bien qu'un terme est équivalent au terme en lequel il se réduit.

Ce principe est appelé *transparence référentielle*.

Induction structurelle

Le principe d'induction structurelle est analogue à l'induction naturelle :

Dans le cas des listes, il a la forme suivante :

Pour prouver une propriété $P(xs)$ pour toutes les listes xs ,

1. montrer que $P(\text{List}())$ est vrai (cas de base),
2. pour une liste xs et un élément x quelconques, montrer que :
si $P(xs)$ est vrai, alors $P(x :: xs)$ l'est aussi
(étape d'induction).

Exemple

Nous allons montrer que $(xs ::: ys) ::: zs = xs ::: (ys ::: zs)$ par induction structurelle sur xs .

Cas de base: $List()$

Pour le côté gauche on a :

$$\begin{aligned} & (List() ::: ys) ::: zs \\ = & ys ::: zs && \text{(par la première clause de :::)} \end{aligned}$$

Pour le côté droit on a :

$$\begin{aligned} & List() ::: (ys ::: zs) \\ = & ys ::: zs && \text{(par la première clause de :::)} \end{aligned}$$

Ce cas est donc établi.

Etape d'induction: $x :: xs$

Pour le côté gauche on a :

$$\begin{aligned} & ((x :: xs) ::: ys) ::: zs \\ = & (x :: (xs ::: ys)) ::: zs && \text{(par la seconde clause de :::)} \\ = & x :: ((xs ::: ys) ::: zs) && \text{(par la seconde clause de :::)} \\ = & x :: (xs ::: (ys ::: zs)) && \text{(par hypothèse d'induction)} \end{aligned}$$

Pour le côté droit on a :

$$\begin{aligned} & (x :: xs) ::: (ys ::: zs) \\ = & x :: (xs ::: (ys ::: zs)) && \text{(par la seconde clause de :::)} \end{aligned}$$

Si bien que ce cas-ci (et avec lui la propriété) est établi.

Exercice : Montrez par induction sur xs que $xs ::: List() = xs$.

Exemple (2)

A titre d'exemple plus difficile, considérons la fonction

```
abstract class List[a] { ...
  def reverse: List[a] = this match {
    case List() => List() /* 1ère clause */
    case x :: xs => xs.reverse :: List(x) /* 2ième clause */
  }
}
```

On aimerait prouver la proposition suivante

$$xs.reverse.reverse = xs .$$

On procède par induction sur xs . Le cas de base est facile à établir :

$$\begin{aligned} & List().reverse.reverse \\ = & List().reverse && \text{(par la 1ère clause de reverse)} \\ = & List() && \text{(par la 1ère clause de reverse)} \end{aligned}$$

Pour l'étape d'induction on essaie :

$$\begin{aligned} & (x :: xs).reverse.reverse \\ = & (xs.reverse :: List(x)).reverse && \text{(par la 2ième clause de reverse)} \end{aligned}$$

On ne peut rien faire de plus avec cette expression, on se tourne donc vers le membre droit :

$$\begin{aligned} & x :: xs \\ = & x :: xs.reverse.reverse && \text{(par hypothèse d'induction)} \end{aligned}$$

Les deux côtés se sont simplifiés en des expressions différentes.

On doit donc encore montrer que

$$(xs.reverse :: List(x)).reverse = x :: xs.reverse.reverse$$

Essayer de le prouver directement par induction ne marche pas.

On doit plutôt essayer de *généraliser* l'équation :

$$(ys :: List(x)).reverse = x :: ys.reverse$$

Cette équation peut être prouvée par un second argument d'induction sur ys . (Voir tableau).

Exercice : Est-il vrai que $(xs \text{ drop } m) \text{ at } n = xs \text{ at } (m + n)$ pour tous entiers m, n et toute liste xs ?

Induction structurelle sur les arbres

L'induction structurelle ne se limite pas aux listes ; elle s'applique à n'importe quelle structure d'arbre.

Le principe général d'induction est le suivant :

Pour montrer la propriété $P(t)$ pour tous les arbres d'un certain type,

- montrer $P(l)$ pour toutes les feuilles l de l'arbre,
- pour chaque noeud interne t avec sous-arbres s_1, \dots, s_n , montrer que $P(s_1) \wedge \dots \wedge P(s_n) \Rightarrow P(t)$.

Exemple : Rappelons notre définition de *IntSet* avec les opérations *contains* et *incl* :

```
abstract class IntSet {  
  def incl(x: Int): IntSet  
  def contains(x: Int): Boolean  
}
```

```

case class Empty extends IntSet {
  def contains(x: Int): Boolean = false
  def incl(x: Int): IntSet = NonEmpty(x, Empty, Empty)
}
case class NonEmpty(elem: Int, left: Set, right: Set) extends IntSet {
  def contains(x: Int): Boolean =
    if (x < elem) left contains x
    else if (x > elem) right contains x
    else true
  def incl(x: Int): IntSet =
    if (x < elem) NonEmpty(elem, left incl x, right)
    else if (x > elem) NonEmpty(elem, left, right incl x)
    else this
}

```

(Avec ajouts de **case** pour pouvoir utiliser les fonctions de construction au lieu de **new**).

Que signifie prouver la correction de cette implantation ?

Les lois de IntSet

Un moyen pour définir et montrer la correction d'une implantation consiste à prouver des lois qu'elle respecte.

Dans le cas de *IntSet*, nous avons les trois lois suivantes :

Pour tout ensemble *s*, et éléments *x* et *y* :

<i>Empty</i> contains <i>x</i>	=	false	
(<i>s</i> incl <i>x</i>) contains <i>x</i>	=	true	
(<i>s</i> incl <i>x</i>) contains <i>y</i>	=	<i>s</i> contains <i>y</i>	si <i>x</i> ≠ <i>y</i>

(En fait, on peut montrer que ces lois caractérisent complètement le type de donnée désiré).

Comment peut-on prouver ces lois ?

Proposition 1: *Empty* contains *x* = **false**.

Preuve : D'après la définition de *contains* dans *Empty*.

Proposition 2: $(s \text{ incl } x)$ contains $x = \mathbf{true}$

Preuve :

Cas de base: *Empty*

$(\text{Empty incl } x)$ contains x
= (d'après définition de incl dans *Empty*)
 $\text{NonEmpty}(x, \text{Empty}, \text{Empty})$ contains x
= (d'après la définition de contains dans *NonEmpty*)
true

Etape d'induction: $\text{NonEmpty}(x, l, r)$

$(\text{NonEmpty}(x, l, r) \text{ incl } x)$ contains x
= (d'après la définition de incl dans *NonEmpty*)
 $\text{NonEmpty}(x, l, r)$ contains x
= (d'après la définition de contains dans *NonEmpty*)
true

Etape d'induction: $\text{NonEmpty}(y, l, r)$ avec $y < x$

$(\text{NonEmpty}(y, l, r) \text{ incl } x)$ contains x
= (d'après la définition de incl dans *NonEmpty*)
 $\text{NonEmpty}(y, l, r \text{ incl } x)$ contains x
= (d'après la définition de contains dans *NonEmpty*)
 $(r \text{ incl } x)$ contains x
= (par hypothèse d'induction)
true

Etape d'induction: $\text{NonEmpty}(y, l, r)$ avec $y > x$ est analogue.

Proposition 3 : Si $x \neq y$ alors $xs \text{ incl } y$ contains $x = xs$ contains x .

Preuve : Voir tableau.

Exercice

Supposons qu'on ajoute une fonction *union* à *IntSet* :

```
class IntSet { ...
  def union(other: IntSet): IntSet
}
class Expty extends IntSet { ...
  def union(other: IntSet) = other
}
class NonEmpty(x: Int, l: IntSet, r: IntSet) extends IntSet { ...
  def union(other: IntSet): IntSet = l union r union other incl x
}
```

La correction de *union* peut alors se traduire par la loi suivante :

Proposition 4: $(xs \text{ union } ys) \text{ contains } x = xs \text{ contains } x \mid\mid ys \text{ contains } x$.

Est-ce vrai ? Quelle hypothèse manque ? Trouvez un contre-exemple.

Montrez la proposition 4 en utilisant une induction structurelle sur *xs*.