Tuples

Scala a une syntaxe spéciale pour les couples et les tuples en général.

Un couple de valeurs x et y s'écrit (x, y).

Si x est de type T et y de type U, alors (x, y) a le type "tuple" (T, U).

Plus généralement, si $x_i : T_i$, alors le tuple $(x_1, ..., x_n)$ a le type $(T_1, ..., T_n)$.

Les tuples et les listes agrègent, l'un comme l'autre, les données, mais il y a deux différences importantes.

- Les éléments d'un tuple peuvent être de types différents, alors que les éléments d'une liste sont tous du même type.
- Le nombre d'éléments d'un tuple est fixé par son type, alors que la longueur d'une liste ne l'est pas.

On accède aux éléments d'un n-tuple par les sélecteurs *_1*, ..., *_n*.

Exemple :

```
> val xy = (2, 3)
val xy : scala.Tuple2 = (2,3)
> xy._1 + xy._2
5
```


La classe Tuple

Comme tous les autres types, les tuples sont représentés en Scala par des classes.

Par exemple le type (T_1, T_2), des tuples à 2 éléments (couples), est représenté par le type classe *scala.Tuple2*$[T_1, T_2]$.

La classe *Tuple2* est définie comme suit.

```
package scala with {
  class Tuple2[a, b](x: a, y: b) with {
    def _1: a = x
    def _2: b = y
    override def toString = "(" + _1 + "," + _2 + ")"
  }
}
```


Explications :

- La clause ***package*** joue un rôle similaire à celle de Java : elle place *Tuple2* dans le paquetage *scala*.
- La classe *Tuple2* est paramétrée par les paramètres de type $[a, b]$ et les paramètres de valeurs (x, y).
- Elle définit les *fonctions d'accès* *_1*, *_2*.
- Elle *redéfinit* aussi la fonction *toString* de *Object*.

Les tuples plus grands peuvent être définis de manière analogue (mais ne font pas partie pour l'instant des classes prédéfinies).

- 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. Cela peut être réalisé par la fonction

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

- Pour chaque entier i, générer la liste des couples $(i, 1), \ldots, (i, i{-}1)$. On peut y arriver par une combination de *range* et *map* :

```
range(1, i-1) map (x => (i, x))
```

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

En rassemblant les morceaux on obtient l'expression suivante :

```
range(1, n)
  .map(i => range(1, i-1).map(x => (i, x)))
  .foldRight(xs: List[(Int, Int)], ys: List[(Int, Int)] => xs ::: ys)([])
```


Traitements imbriqués sur les listes

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

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

Par exemple, si $n = 6$, 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 façon naturelle de faire cela consiste à :

- Générer la suite de tous les couples d'entiers (i, j) tels que $1 \le j < i \le n$.

D'avantage 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(y: a, ys: List[b] =>   ??   )([])

def concatFun[a](xs: List[a], ys: List[a]): List[a] =
  ??  .foldRight(cons[a])(  ??  )

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

Ici, *cons* est prédéfinie dans le fichier *List.scala* par

```
def cons[a](x: a, xs: List[a]): List[a] = x :: xs
```


La fonction *zip*

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

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

Exemple : En utilisant *zip* et *fold*, 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)
  .fold(x: Double, y: Double => x + y)(0)
```

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] with {
  ...
  def flatMap[b](f: (a)List[b]): List[b] = {
    map(f).foldRight(xs: List[b], ys: List[b] ⇒ xs ::: ys)([])
  }
}
```

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

```
range(1, n)
  .flatMap(i ⇒ range(1, i−1).map(x ⇒ (i, x)))
```

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

Raisonner sur les listes

Rappelons l'opération de concaténation pour les listes :

```
class List[a] with {
  ...
  def ::: (that: List[a]) =
    if (isEmpty) that
    else head :: (tail ::: that)
}
```

On aimerait vérifier que la concaténation est associative, et qu'elle a la liste vide [] comme élément neutre à gauche et à droite.

$$
\begin{array}{lcl}
(xs ::: ys) ::: zs & = & xs ::: (ys ::: zs) \\
xs ::: [] & = & xs \quad = \quad [] ::: xs
\end{array}
$$

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 nombres $n \geq b$,

1. Montrer qu'on a $P(b)$ (cas de base).
2. Pour tout $n \geq b$ montrer que :

   si on a $P(n)$, alors on a aussi $P(n+1)$

   (étape d'induction).

Exemple: Etant donné

```
def factorial(n: Int): Int =
  if (n == 0) 1
  else n * factorial(n−1)
```

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

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

Cas 4 est établi par simple calcul de $factorial(4) = 24$ et $2^4 = 16$.

Cas n+1 On a pour $n \geq 4$:

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

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

Ca 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*.

Exemple (2)

A titre d'exemple plus difficile, considérons la fonction

```
abstract class List[a] with {
  ...
  def reverse: List[a] =
    if (isEmpty) []
    else tail.reverse ::: [head]
}
```

On aimerait prouver la validité de la proposition suivante

$$xs.reverse.reverse \;=\; xs \quad .$$

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

$$\begin{array}{ll} & [\,].reverse.reverse \\ = & \text{(par la première clause de } reverse\text{)} \\ & [\,].reverse \\ = & \text{(par la première clause de } reverse\text{)} \\ & [\,] \end{array}$$

$\boxed{\textbf{Cas } x :: xs}$

Pour le côté gauche on a :

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

Pour le côté droit on a :

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

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

Exercice : Montrer par induction sur xs que $xs ::: [\,] \;=\; xs$.

Exemple

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

$\boxed{\textbf{Cas } [\,]}$ Pour le côté gauche on a :

$$\begin{array}{ll} & ([\,] ::: ys) ::: zs \\ = & \text{(par la première clause de :::)} \\ & ys ::: zs \end{array}$$

Pour le côté droit on a :

$$\begin{array}{ll} & [\,] ::: (ys ::: zs) \\ = & \text{(par la première clause de :::)} \\ & ys ::: zs \end{array}$$

Ce cas est donc établi.

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 $P([\,])$ 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).

Pour l'étape d'induction on essaie :

$$\begin{aligned} & (x :: xs).reverse.reverse \\ = \quad & \textit{(par la seconde clause de } \mathrm{reverse}) \\ & (xs.reverse ::: [x]).reverse \end{aligned}$$

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

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

Les deux côtés se sont simplifiés en des expressions différentes.

On doit donc encore montrer que

$$(xs.reverse ::: [x]).reverse \;=\; x :: xs.reverse.reverse$$

Essayer de le prouver directement par induction ne marche pas.

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

$$(ys ::: [x]).reverse \;=\; x :: ys.reverse$$

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

Exercice : Est-il vrai que $(xs\ drop\ m)\ at\ n = xs\ at\ (m+n)$ pour tous nombres naturels 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 de l'arbre.
- Pour chaque noeud interne t avec sous-arbres $s_1, \ldots, s_n$, montrer que $P(s_1) \wedge \ldots \wedge P(s_n) \Rightarrow P(t)$.

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

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



```
class Empty extends IntSet with {
  def contains(x: Int): Boolean = False
  def incl(x: Int): IntSet = NonEmpty(x, Empty, Empty)
}
class NonEmpty(elem: Int, left: Set, right: Set) extends IntSet with {
  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
}
```

Que signifie prouver la correction de cette implantation ?

Exercice

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

```
class IntSet with {
  ...
  def union(other: IntSet): IntSet
}
class Expty extends IntSet with {
  ...
  def union(other: IntSet) = other
}
class NonEmpty(x: Int, l: IntSet, r: IntSet) extends IntSet with {
  ...
  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 : $(xs\ union\ ys)\ contains\ x\ =\ xs\ contains\ x\ ||\ ys\ contains\ x$.

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

$\boxed{\text{Cas } NonEmpty(y, l, r) \text{ avec } y < x}$

$$\begin{array}{cl} & (NonEmpty(y, l, r)\ incl\ x)\ contains\ x \\ = & (\text{d'après la définition de } incl \text{ dans } NonEmpty) \\ & NonEmpty(y, l, r\ incl\ x)\ contains\ x \\ = & (\text{d'après la définition de } contains \text{ dans } NonEmpty) \\ & (r\ incl\ x)\ contains\ x \\ = & (\text{par hypothèse d'induction}) \\ & True \end{array}$$

$\boxed{\text{Cas } NonEmpty(y, l, r) \text{ avec } y > x}$ Idem.

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

Preuve : Voir le tableau.

Proposition 2 : $(xs\ incl\ x)\ contains\ x = True$

Preuve :

$\boxed{\text{Cas } Empty}$

$$\begin{array}{cl} & (Empty\ incl\ x)\ contains\ x \\ = & (\text{d'après définition de } incl \text{ dans } Empty) \\ & NonEmpty(x, Empty, Empty)\ contains\ x \\ = & (\text{d'après la définition de } contains \text{ dans } NonEmpty) \\ & True \end{array}$$

$\boxed{\text{Cas } NonEmpty(x, l, r)}$

$$\begin{array}{cl} & (NonEmpty(x, l, r)\ incl\ x)\ contains\ x \\ = & (\text{d'après la définition de } incl \text{ dans } NonEmpty) \\ & NonEmpty(x, l, r)\ contains\ x \\ = & (\text{d'après la définition de } contains \text{ dans } Empty) \\ & True \end{array}$$

Les lois de IntSet

Une manière de définir et prouver la correction d'une implantation consiste à prouver des lois qu'elle respecte.

Dans le cas de *IntSet*, les trois lois suivantes en sont un exemple.

Pour tout ensemble s, et éléments x, y :

$$\begin{array}{lcll} Empty\ contains\ x & = & False & \\ (s\ incl\ x)\ contains\ x & = & True & \\ (s\ incl\ x)\ contains\ y & = & s\ contains\ y & \quad si\ x \neq y \end{array}$$

(En fait, on peut montrer que ces lois caractérisent complétement 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*.