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).

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)
```

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.

$$\textbf{\textit{def}}\ \mathit{mapFun}\,[\mathrm{a},\ \mathrm{b}\,]\,(\mathit{xs}\colon \mathit{List}\,[\mathrm{a}\,],\ f\colon\ (\mathrm{a}\,)\mathrm{b}\,)\colon \mathit{List}\,[\mathrm{b}\,] =$$
$$\quad \mathit{xs.foldRight}\,(y\colon \mathrm{a},\ \mathit{ys}\colon \mathit{List}\,[\mathrm{b}\,] \Rightarrow \quad ?? \quad)\,(\,[\,]\,)$$

$$\textbf{\textit{def}}\ \mathit{concatFun}\,[\mathrm{a}\,]\,(\mathit{xs}\colon \mathit{List}\,[\mathrm{a}\,],\ \mathit{ys}\colon \mathit{List}\,[\mathrm{a}\,]\,)\colon \mathit{List}\,[\mathrm{a}\,] =$$
$$\quad ??\ \ .\mathit{foldRight}\,(\mathit{cons}\,[\mathrm{a}\,]\,)\,(\quad ?? \quad)$$

$$\textbf{\textit{def}}\ \mathit{lengthFun}\,[\mathrm{a}\,]\,(\mathit{xs}\colon \mathit{List}\,[\mathrm{a}\,]\,)\colon \mathit{Int} =$$
$$\quad \mathit{xs.foldRight}\,(\quad ?? \quad)\,(\mathit{0}\,)$$

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

$$\textbf{\textit{def}}\ \mathit{cons}\,[\mathrm{a}\,]\,(x\colon \mathrm{a},\ \mathit{xs}\colon \mathit{List}\,[\mathrm{a}\,]\,)\colon \mathit{List}\,[\mathrm{a}\,] = x :: \mathit{xs}$$

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 $1 \leq j < i \leq n$ 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 $1 \leq j < i \leq n$.

- 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. 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)([])
```

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 :

$$
\begin{array}{l}
\textbf{\textit{class}}\ \mathit{List}\,[\mathrm{a}]\ \textbf{\textit{with}}\ \{ \\
\quad \ldots \\
\quad \textbf{\textit{def}}\ :::\ (\mathit{that}: \mathit{List}\,[\mathrm{a}]) = \\
\qquad \textbf{\textit{if}}\ (\mathit{isEmpty})\ \mathit{that} \\
\qquad \textbf{\textit{else}}\ \mathit{head}\ ::\ (\mathit{tail}\ :::\ \mathit{that}) \\
\}
\end{array}
$$

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

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

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 $n \geq 4$,

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

$\boxed{\textbf{Cas } \mathit{4}}$ est établi par simple calcul de $\mathit{factorial}(\mathit{4}) = \mathit{24}$ et $\mathit{2}^{\mathit{4}} = \mathit{16}$.

$\boxed{\textbf{Cas } n\mathit{+1}}$ On a pour $n \geq \mathit{4}$:

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

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*.

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).

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 :

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

Pour le côté droit on a :

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

Ce cas est donc établi.

Cas $x :: xs$

Pour le côté gauche on a :

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

Pour le côté droit on a :

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

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

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

Exemple (2)

A titre d'exemple plus difficile, considérons la fonction

$$
\begin{array}{l}
\textbf{\textit{abstract class}}\ \mathit{List}[\mathrm{a}]\ \textbf{\textit{with}}\ \{ \\
\quad \ldots \\
\quad \textbf{\textit{def}}\ \mathit{reverse}: \mathit{List}[\mathrm{a}] = \\
\qquad \textbf{\textit{if}}\ (\mathit{isEmpty})\ [\,] \\
\qquad \textbf{\textit{else}}\ \mathit{tail.reverse} ::: [\mathit{head}] \\
\}
\end{array}
$$

On aimerait prouver la validité de la proposition suivante

$$
\mathit{xs.reverse.reverse} \;=\; \mathit{xs} \quad .
$$

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

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

Pour l'étape d'induction on essaie :

$$\begin{array}{ll} & (x :: xs).reverse.reverse \\ = & (\textit{par la seconde clause de } \mathit{reverse}) \\ & (xs.reverse ::: [x]).reverse \end{array}$$

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

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

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, ..., s_n, montrer que $P(s_1) \wedge ... \wedge P(s_n) \Rightarrow P(t)$.

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 ?

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 :

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

(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*.

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

Preuve :

$\boxed{\textbf{Cas}\ Empty}$

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

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

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

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

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

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

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

Preuve : Voir le tableau.

Exercice

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

$$
\begin{array}{l}
\textbf{\textit{class}}\ \mathit{IntSet}\ \textbf{\textit{with}}\ \{\\
\quad \ldots\\
\quad \textbf{\textit{def}}\ \mathrm{union}\,(\mathit{other}\colon \mathit{IntSet}\,)\colon \mathit{IntSet}\\
\}\\
\textbf{\textit{class}}\ \mathit{Expty}\ \textbf{\textit{extends}}\ \mathit{IntSet}\ \textbf{\textit{with}}\ \{\\
\quad \ldots\\
\quad \textbf{\textit{def}}\ \mathrm{union}\,(\mathit{other}\colon \mathit{IntSet}\,) = \mathit{other}\\
\}\\
\textbf{\textit{class}}\ \mathit{NonEmpty}\,(x\colon \mathit{Int},\ l\colon \mathit{IntSet},\ r\colon \mathit{IntSet}\,)\ \textbf{\textit{extends}}\ \mathit{IntSet}\ \textbf{\textit{with}}\ \{\\
\quad \ldots\\
\quad \textbf{\textit{def}}\ \mathrm{union}\,(\mathit{other}\colon \mathit{IntSet}\,)\colon \mathit{IntSet} = l\ \mathit{union}\ r\ \mathit{union}\ \mathit{other}\ \mathit{incl}\ x\\
\}
\end{array}
$$

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

Proposition 4 : $(\mathit{xs}\ \mathit{union}\ \mathit{ys})\ \mathit{contains}\ x \;=\; \mathit{xs}\ \mathit{contains}\ x\ ||\ \mathit{ys}\ \mathit{contains}\ x.$

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