Semaine 10 : Calcul avec les flots

Les deux dernières semaines nous avons introduit les affectations et les objets à états.

Nous avons vu comment modéliser un changement d'état dans un objet réel par des affectations de variable pendant le calcul.

Cela signifie que les changements temporels du monde réel sont modélisés par des changements temporels dans l'exécution du programme (étirés ou compressés, mais l'ordre relatif reste le même).

Y a-t-il une autre façon de faire ?

Peut-on modéliser un comportement changeant dans le monde réel par des fonctions sans mutation ?

En mathématiques : Bien sûr !

Une quantité changeant avec le temps se modélise simplement par une fonction $f(t)$ avec un paramètre temporel t.

On peut faire la même chose dans un calcul. Voici le principe :

- Au lieu d'affecter à une variable les valeurs successives, on construit une liste avec les valeurs successives de la variable.
- Par ex. ***var*** x: T devient **val** x: $Stream[T]$.
- Ici, les flots sont des séquences de valeurs ressemblant aux listes.

Les flots sont des listes retardées

Un avantage du modèle par flots est que nous pouvons utiliser les fonctions d'ordre supérieur de manipulation de listes.

Cela simplifie souvent le calcul.

Exemple : La manière impérative de calculer la somme de tous les nombres premiers d'un intervalle s'écrit comme suit.

```
def sumPrimes(lo: Int, hi: Int): Int = {
  var i = lo;
  var acc = 0;
  while (i < hi) {
    if (isPrime(i)) acc = acc + i;
    i = i + 1;
  }
  acc
}
```

Notez que la variable i prend toutes les valeurs de l'intervalle $[lo\ ..\ hi]$.

Une manière plus fonctionnelle est de représenter la liste des valeurs de la variable i directement comme *range*(*lo*, *hi*). La fonction peut alors se réécrire comme suit.

```
def sumPrimes(lo: Int, hi: Int) =
  sum(range(lo, hi) filter isPrime);
```

Pas d'hésitation pour désigner le programme le plus court et le plus clair !

Cependant le programme fonctionnel est aussi considérablement moins efficace car il construit une liste de tous les nombres de l'intervalle, et ensuite une autre pour les nombres premiers.

Encore pire du point de vue de l'efficacité est l'exemple suivant :

Pour trouver le deuxième nombre premier entre *1000* et *10000* :

```
range(1000, 10000) filter isPrime at 1
```

Ici, la liste de tous les nombres entre *1000* et *10000* est construite.

La plus grande partie de cette liste n'est jamais inspectée !

Utiliser les flots

On crée les flots en utilisant les deux constructeurs *EmptyStream* et *ConsStream*.

Exemple :

```
val ConsStream(1, ConsStream(2, EmptyStream))  // le flot [1, 2]

def streamRange(lo: Int, hi: Int): Stream[Int] =
  if (lo > hi) EmptyStream
  else ConsStream(lo, streamRange(lo + 1, hi));
```

Même si *streamRange* ressemble à *range*, le comportement de son exécution est complétement différent :

- *streamRange* retourne immédiatement un objet *Stream* dont le premier élément est *lo*.
- Les autres éléments ne sont calculés que quand ils sont demandés par un appel de la méthode *tail* (ce qui peut ne jamais arriver).

Evaluation retardée

Cependant, on peut obtenir une exécution efficace pour des exemples comme ceux-ci en utilisant une astuce :

> Eviter de calculer la queue d'une séquence à moins qu'elle ne soit effectivement nécessaire pour le calcul.

On définit une nouvelle classe pour de telles séquences, qu'on appelle *Stream*.

Concaténer une liste ou un élément à un flot

Les flots ne supportent pas les méthodes :: et :::, car elles requièrent que leur opérande droite soit évaluée en premier.

L'opérateur infixe :: est remplacé par la fonction constructeur *ConsStream*. Par ex. à la place de *x* :: *xs* on écrit *ConsStream* (*x*, *xs*).

L'opérateur infixe ::: est remplacé par l'opérateur *append*. Par ex. à la place de *xs* ::: *ys* on écrit *xs append ys*.

Ici, on définit *append* comme d'habitude :

```
def append (def rest : Stream [a] ): Stream [a] =
  if (isEmpty) rest
  else ConsStream (head, tail.append (rest) );
```


On accède aux flots exactement comme aux listes, en utilisant *isEmpty*, *head* et *tail*.

Exemple : Pour afficher tous les éléments d'un flot :

```
print (xs: Stream [a] ): Unit =
  if (xs.isEmpty) () else { System.out.println (xs.head); print (xs.tail) }
```

De plus, les flots supportent presque toutes les méthodes sur les listes.

Exemple : Pour trouver le 2ième nombre premier entre *1000* et *10000* :

```
streamRange (1000, 10000) filter isPrime at 1
```

Implantation de *Stream*

L'implantation de la classe *Stream* est très similaire à l'implantation de la classe *List*.

Il y a une classe de base abstraite *Stream*, définie comme suit.

```
abstract class Stream[a] with {
  abstract def isEmpty: Boolean;
  abstract def head: a;
  abstract def tail: Stream[a];
  ...
}
```

Il y a aussi d'autres méthodes dans *Stream*, telles que celles décrites ci-dessus.

Une grande partie de l'interface *Stream* est présentée ci-dessous.


```
abstract class Stream[a] with {
  abstract def isEmpty: Boolean;
  abstract def head: a;
  abstract def tail: Stream[a];
  def append(that: Stream[a]): Stream[a];
  def length: Int;
  def take(n: Int): Stream[a];
  def drop(n: Int): Stream[a];
  def at(n: Int): a;
  def map[b](f: (a)b): Stream[b];
  def foreach(f: (a)Unit): Unit;
  def filter(p: (a)Boolean): Stream[a];
  def forall(p: (a)Boolean): Boolean;
  def exists(p: (a)Boolean): Boolean;
  def fold[b](op: (b, a)b)(z: b): b;
  def foldRight[b](op: (a, b)b)(z: b): b;
  def flatMap[b](f: (a)Stream[b]): Stream[b];
  def zip[b](that: Stream[b]): Stream[(a, b)];
  def print: Unit;
}
```

La seule différence importante entre les implantations de *List* et de *Stream* concerne le statut de *tl*, le deuxième paramètre de *ConsStream*.

Ce paramètre est précédé de ***def***; c'est pourquoi l'argument correspondant ne sera pas évalué lors de son passage à *ConsStream*.

Au contraire, un tel argument sera évalué à chaque fois que *tl* sera appelée sur l'objet *ConsStream*.

Flots concrets

Il y a aussi deux sous-classes concrètes, *EmptyStream* et *ConsStream*. On peut les définir comme suit.

```
final class EmptyStream[c] extends Stream[c] with {
    def isEmpty = True;
    def head: c = error("head of empty stream");
    def tail: Stream[c] = error("tail of empty stream");
    override def toString(): String = "EmptyStream";
}
final class ConsStream[b](hd: b, def tl: Stream[b]) extends Stream[b]
    with {
        def isEmpty = False;
        def head = hd;
        def tail = tl;
        override def toString(): String = "ConsStream(" + hd + ", ?)";
}
```

L'implantation des flots en action

Pour voir que l'implantaion des flots se comporte réellement efficacement, nous observons la trace d'exécution de l'exemple du 2ième nombre premier entre *1000* et *10000*.

(Nous sommes de retour à la programmation fonctionnelle pure, nous pouvons donc utiliser le modèle par substitution simple).

```
  streamRange(1000, 10000).filter(isPrime).at(1)
→ par expansion de streamRange
  (if (1000 > 10000) EmptyStream
  else ConsStream(1000, streamRange(1000 + 1, 10000)))
  .filter(isPrime).at(1)
→ par évaluation du if
  ConsStream(1000, streamRange(1000 + 1, 10000)).filter(isPrime).at(1)
→ ...
```


Evaluation paresseusse

L'implantation présentée a un problème d'efficacité potentiel : si *tl* est appelée plusieurs fois sur un objet *ConsStream* la queue du flot sera évaluée plusieurs fois.

Ce problème peut être évité en stockant le résultat de la première évaluation de *tl* et en réutilisant le résultat stocké à chaque nouvel appel à *tl*.

Exercice : Ecrire une nouvelle version de *ConsStream* qui implante cette amélioration.

...

→ *par évaluation de* tail

streamRange(1001, 10000).filter(isPrime).at(1)

Le processus continue ainsi jusqu'à atteindre

streamRange(1009, 10000).filter(isPrime).at(1)

→ *par évaluation de* filter,

en notant C_2 = *ConsStream(1009, streamRange(1009 + 1, 10000))*
ConsStream(1009, C_2*.tail.filter(isPrime)).at(1)*

→ *par évaluation de* at

C_2*.tail.filter(isPrime)).at(0)*

→ *par évaluation de* at, *expansion de* C_2

streamRange(1010, 10000).filter(isPrime).at(0)

Le processus continue encore jusqu'à :

...

→ *par expansion de* filter,
en notant C_1 = *ConsStream(1000, streamRange(1000 + 1, 10000))*
(**if** (C_1.isEmpty) ***this***
else if (isPrime(C_1.head)) ConsStream(C_1.head, C_1.tail.filter(isPrime))
else C_1.tail.filter(isPrime)).at(1)

→ *par évaluation du* if

(**if** (isPrime(C_1.head)) ConsStream(C_1.head, C_1.tail.filter(isPrime))
else C_1.tail.filter(isPrime)).at(1)

→ *par évaluation de* head

(**if** (isPrime(1000)) ConsStream(C_1.head, C_1.tail.filter(isPrime))
else C_1.tail.filter(isPrime)).at(1)

→ *par évaluation de* isPrime

(**if** (False) ConsStream(C_1.head, C_1.tail.filter(isPrime))
else C_1.tail.filter(isPrime)).at(1);

→ *par évalaution du* if

C_1.tail.filter(isPrime).at(1);

Flots infinis

Nous avons vu que les éléments d'un flot (sauf le premier) ne sont évalués que s'ils sont requis par le calcul.

Cela rend possible de définir aussi des flots infinis !

Par exemple, voici le flot de tous les entiers, ou tous les entiers à partir d'un nombre n.

```
def from (n: Int): Stream[Int] = ConsStream(n, from(n+1));
val integers = from(0);
```

En utilisant les entiers, on peut aussi définir d'autres flots infinis, comme celui des multiples de 4 :

```
val multiplesOfFour = integers map (x => x * 4)
```

Ou bien le flot des nombres non divisibles par 7 :

```
val noSevens = integers filter (x => x % 7 != 0)
```


streamRange(1013, 10000).filter(isPrime).at(0)

→ *par évaluation de filter,*

en notant C_3 = ConsStream(1013, streamRange(1013 + 1, 10000))
ConsStream(1013, C_3.tail.filter(isPrime)).at(0)

→ *par évaluation de at*

1013

On voit que seule la partie nécessaire des éléments de la liste a été créée.

Itérations comme processus avec flot

Les flots avec évaluation retardée sont un puissant outil de modélisation, car ils nous permettent de construire un système avec des interfaces de modules différentes que si on utilisait des variables d'état.

Par exemple, il est possible de penser à une série temporelle entière comme centre d'intérêt, plutôt qu'à la valeur de la variable à un moment précis.

En particulier, des itérations complètes de processus peuvent être transformées en objets qui se manipulent eux-même.

Un exemple est la fonction racine carrée présentée dans le cours 1.

Cette fonction calculait les racines carrées en appliquant itérativement la fonction :

```
def improve(guess: Double, x: Double) = (guess + x / guess) / 2;
```


Un flot infini plus intéressant est celui des nombres premiers.

On va construire ce flot en utilisant la technique du *crible d'Erathostène*.

- Commencer avec les entiers à partir de *2*, qui est aussi le premier nombre premier.
- Eliminer de la liste tous les multiples de *2*.
- Le premier élément de la liste restante est *3*, qui est premier.
- Eliminer de la liste tous les multiples de *3*.
- Continuer de la même façon. A chaque pas, le premier nombre de la liste est premier, et on élimine tous ses multiples.

Ce processus est défini par la fonction suivante.

```
def sieve(s: Stream[Int]): Stream[Int] =
  ConsStream(s.head, sieve(s.tail filter (x => x % s.head != 0)));

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

Maintenant pour trouver la liste de tous les nombres premiers, afficher simplement la liste (ca va prendre du temps (−;) : *primes.print*

Un autre exemple calcule la valeur de π, en se basant sur la série alternante

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots$$

On commence par prendre le flot des éléments à sommer de la série.

```
def piSummands(n: Int): Stream[Int] =
  ConsStream(1.0/n, piSummands(n + 2) map (x => -x));
```

On prend ensuite le flot des *sommes partielles* de toutes ces éléments, et on le multiplie par 4:

```
val piStream = partialSums(piSummands(1)) map (x => x * 4)
```

Cela nous donne un flot d'approximations de π, mais la vitesse de convergence est lente :

On peut maintenant calculer le flot infini des approximations de la racine carrée :

```
def sqrtStream(x: Double): Stream[Double] = {
  val guesses = ConsStream(1, guesses map (guess => improve(guess, x)));
  guesses
}
```

On a alors :

```
> sqrtStream print
1
1.5
1.416666666665
...
```

Transformer les flots

Jusqu'à maintenant, notre utilisation des flots n'était pas si différente de la mise à jour de variables d'états.

Mais les flots nous permettent d'utiliser des astuces intéressantes.

Par exemple, on peut utiliser un *accélérateur de suite* qui convertit une suite d'approximations en une autre qui converge plus rapidement.

Un tel accélérateur, dû à Leonhard Euler, un mathématicien suisse du 18ième siècle, marche bien avec les suites qui sont les sommes partielles d'une série alternante.

Avec l'accélérateur d'Euler, si la série originale a pour termes s_i, la série transformée t a pour termes

$$t_i \quad = \quad s_{i+1} - \frac{(s_{i+1}-s_i)^2}{s_{i-1}-2s_i+s_{i+1}}$$

> *piStream print*
4.0, 2.666666666666667, 3.466666666666667, 2.8952380952380956,
3.3396825396825, 2.976046176046176, 3.2837384837384,
3.017071817071817, ...

Exercice : La fonction *partialSums* calcule pour un flot donné *xs* un nouveau flot $s_1, s_2, s_3, \ldots,$ où chaque s_i est donné par :

$$s_i \quad = \quad \sum_{j=1}^{i} xs_j$$

Compléter l'implantation suivante de *partialSums*.

```
def partialSums(s: Stream[Double]): Stream[Double] =
  ConsStream(s.head, partialSums(s.tail) map ?? );
```

Tableaux

Encore mieux, on peut accélérer la série accélérée, et récursivement l'accélérer encore, etc.

Plus précisément, on peut créer un flot de flots (appelé un *tableau*), dans lequel chaque flot est la transformée du précédent :

```
def tableau(transform: (Stream[Double])Stream[Double],
            s: Stream[Double]): Stream[Stream[Double]] =
  ConsStream(s, tableau(transform, transform(s)));
```

Pour la transformée d'Euler, le tableau a la forme suivante :

$$\begin{array}{cccccc} s_{00} & s_{01} & s_{02} & s_{03} & s_{04} & \dots \\ & s_{10} & s_{11} & s_{12} & s_{13} & \dots \\ & & s_{20} & s_{21} & s_{22} & \dots \\ & & & \dots & & \end{array}$$

Donc, si la série originale est un flot de valeurs, sa transformée d'Euler est donnée par :

```
def eulerTransform(s: Stream[Double]): Stream[Double] = {
  val s0 = s at 0;
  val s1 = s at 1;
  val s2 = s at 2;
  ConsStream(
    s2 - (s2 - s1) * (s2 - s1) / (s0 - 2 * s1 + s2),
    eulerTransform(s.tail));
}
```

On peut démontrer l'accélération d'Euler avec notre suite d'approximations de *pi*.

```
> eulerTransform(piStream) print
3.166666666666667, 3.1333333333333337, 3.1452380952380956,
3.13968253968253 96, 3.142712842712843, 3.140881340881341,
3.142071817071817, 3.1412548236077646, ...
```

Itérateurs

Les flots évitent la construction des éléments d'une liste non utilisés.

Mais ils s'accompagnent encore du surcoût lié à la construction de listes.

Une version impérative plus légère sont les *itérateurs*.

Un itérateur représente une suite d'éléments qui sont évalués à la demande.

Mais plutôt que de construire cette séquence explicitement, il retourne successivement les éléments avec une fonction *next*.

Les deux plus importantes fonctions définies par un itérateur sont données ici :

```
class Iterator[a] with {
  abstract def hasNext:
  abstract def next: a
}
```


Diagonalisation

Pour finir, on peut former une suite qui prend le premier terme de chaque ligne du tableau :

```
def accelerate(transform: (Stream[Double])Stream[Double],
               s: Stream[Double]): Stream[Double] =
  tableau(transform, s) map (x => x.head);
```

On peut démontrer cette sorte de ¡¡super-accélération¿¿ de la suite de π :

```
> accelerate(eulerTransform, piStream)
4.0, 3.166666666666667, 3.142105263157895, 3.141599357319004,
3.1415927140337785, 3.1415926539752927, 3.141592653591176, 3.141592653589777, ...
```

Bien sûr, on peut implanter ces techniques d'accélération sans utiliser les flots.

Mais la formulation avec flots est particulièrement élégante et concise, car la séquence entière des états est disponible au même instant, en tant que donnée.

hasNext retoune *True* ssi l'itérateur a encore des éléments.

next retourne l'élément suivant de l'itérateur, et en même temps avance l'itérateur d'une unité.

Important : *next* pour les itérateurs et *head* pour les flots ne sont pas identiques : Appelée de façon répétée, *next* retourne successivement les éléments de l'itérateur, alors que *head* retourne toujours le même élément.

Les itérateurs sont aussi présents dans Java ; ils sont un ingrédient essentiel de la bibliothèque *java.util* dédiée aux collections.

Exceptions faites de *head*, *tail*, et *isEmpty*, qui sont remplacées par *next* et *hasNext*, les itérateurs supportent essentiellement les mêmes méthodes que les flots.

En particulier, il y a *length*, *append*, *take* et *drop*, *map* et *flatMap*, *foreach*, *fold* et *foldRight*, *zip*, etc.

Une grande partie de l'interface *Iterator* est donnée ci-dessous.


```
abstract class Iterator[a] with {
    abstract def hasNext: Boolean;
    abstract def next: a;
    def append(that: Iterator[a]): Iterator[a];
    def length: Int;
    def take(n: Int): Iterator[a];
    def drop(n: Int): Iterator[a];
    def at(n: Int): a;
    def map[b](f: (a)b): Iterator[b];
    def foreach(f: (a)Unit): Unit;
    def filter(p: (a)Boolean): Iterator[a];
    def forall(p: (a)Boolean): Boolean;
    def exists(p: (a)Boolean): Boolean;
    def fold[b](op: (b, a)b)(z: b): b;
    def foldRight[b](op: (a, b)b)(z: b): b;
    def flatMap[b](f: (a)Iterator[b]): Iterator[b];
    def zip[b](that: Iterator[b]): Iterator[(a, b)];
    def print: Unit;
}
```

Mais des sous-classes comme celles-ci ne sont pas utilisées si souvent, étant donné que la plupart des itérateurs sont construits directement.

Exemple : Voici l'analogue de *from* pour les itérateurs :

```
def from (n: Int): Iterator[Int] = new Iterator[Int] with {
    var cur = n - 1;
    def hasNext = True;
    def next = { cur = cur + 1; cur }
}
```


Sous-classes de Iterator

Voici deux sous-classes des itérateurs.

La première classe représente l'itérateur vide, qui n'a pas d'éléments.

```
final class EmptyIterator[c] extends Iterator[c] with {
    def isEmpty = True;
    def hasNext = False;
    def next: c = error("next on empty iterator");
}
```

La deuxième classe représente un itérateur qui retourne tous les éléments d'une liste donnée.

```
final class ListIterator[a](xs: List[a]) extends Iterator[a] with {
    var cur: List[a] = xs;
    def hasNext: Boolean = cur.nonEmpty;
    def next: a = { val x = cur.head; cur = cur.tail; x }
}
```

Utiliser les itérateurs

Du code utilisant des fonctions d'ordre supérieur sur les flots peut souvent être converti directement en itérateurs.

Exemple : Calcul des nombres premiers en utilisant un itérateur :

```
from(2) filter isPrime
```

Il est aussi possible de mélanger les itérateurs et les flots :

```
def sieve(it: Iterator[Int]): Stream[Int] = {
  val head = it.next;
  ConsStream(head, sieve(it filter (x => x % head != 0)))
}
sieve(from(2))
```

Mais : Avec les itérateurs il faut faire attention qu'un même itérateur ne soit jamais exécuté plus d'une fois (car la première utilisation va en fait le modifier !)

En particulier, les techniques d'accélération de série telles que la

transformée d'Euler ne peuvent pas être utilisées telles quelles, car elles accèdent à une suite plusieurs fois.

```
def filter(p: (a)Boolean) = new Iterator[a] with {
  private var head: a;
  private var isAhead = False;
  def hasNext: Boolean =
    if (isAhead) True
    else if (outer.hasNext) {
      head = outer.next;
      isAhead = p(head);
      hasNext
    }
    else False;
  def next: a =
    if (hasNext) { isAhead = False; head }
    else error("next on empty iterator");
}
```


Implantation des méthodes d'Iterator

Voici les implantations de quelques méthodes de la classe *Iterator*.

Ici, *outer* indique un identificateur en dehors de la portée de la classe ou du template ***new***).

```
abstract class Iterator[a] with {

  def append(that: Iterator[a]): Iterator[a] = new Iterator[a] with {
    def hasNext = outer.hasNext || that.hasNext;
    def next = if (outer.hasNext) outer.next else that.next;
  }

  def map[b](f: (a)b) = new Iterator[b] with {
    def hasNext: Boolean = outer.hasNext;
    def next: b = f(outer.next);
  }
```

Itérateurs et flots

Les itérateurs et les flots représentent tout deux des séquences et leurs éléments sont évalués à la demande dans les deux cas.

Alors, quelle abstraction est préférable ?

De façon courte, les itérateurs sont les flots ¡¡ du pauvre ¿¿.

Leurs avantages sont :

- Ils peuvent être plus rapides vu qu'aucun objet *ConsStream* n'est construit.
- Ils peuvent être implantés même dans des langages appauvris sans paramètres avec appel par besoin ou fonctions d'ordre supérieur.

Leurs inconvénients sont :

- Une dépendance d'ordre : il faut faire attention à l'ordre dans lequel les opérations *next* sont appelées.
- Pas de réutilisation : une fois qu'un itérateur a avancé, l'ancien état

Exercice : Implanter la méthode

$$\textbf{\textit{def}}\ foreach(f\colon\ (a)Unit)\colon\ Unit$$

qui exécute la fonction f pour tous les éléments de l'itérateur.

Exercice : Implanter une méthode

$$\textbf{\textit{def}}\ dropWhile(p\colon\ (a)Boolean)\colon\ Iterator[a]$$

qui retourne l'itérateur obtenu à partir de l'itérateur courant en effaçant les premiers éléments tant qu'ils vérifient le prédicat p.

est perdu ; si bien qu'on ne peut pas parcourir un itérateur plus d'une fois.