Semaine 10 : Calculer avec les flots

Les deux dernières semaines nous avons introduit les affectations et les objets avec état.

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 (espacés ou rapprochés, mais l'ordre 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 qui ne modifient pas d'état ?

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 contenant les valeurs successives de la variable. Une telle liste (potentiellement infinie) est appelée un *flot* (stream).
- Par exemple, **var** $x\colon T$ devient **val** $x\colon \mathit{Stream}\,[T]$.

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 les algorithmes.

Example: 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
}
```

La variable i va prendre toutes les valeurs de l'intervalle $[lo\,..\,hi{-}1]$.

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

```
def sumPrimes(lo: int, hi: int): int =
  sum(List.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 :

```
List.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 !

Évaluation retardée

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

> Éviter 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*.

Utilisation des flots

Les flots sont créés en utilisant une constante *empty* et un constructeur binaire *cons*. Les deux sont définis dans le module *Stream*. **Example:**

```
val xs = Stream.cons(1, Stream.cons(2, Stream.empty)); // le flot (1, 2)
def range(start: int, end: int): Stream[int] =
  if (start ≥ end) empty
  else cons(start, range(start + 1, end));
```

(En fait, c'est la définition exacte de *range* dans le module *scala.Stream*)

Même si *Stream.range* et *List.range* se ressemblent, leur comportement à l'exécution est complètement différent.

- *Stream.range* retourne immédiatement un objet *Stream* dont le premier élément est *start*.
- Les autres éléments ne sont calculés que lorsqu'ils sont demandés par un appel à la méthode *tail* (ce qui peut ne jamais arriver).

Comme avec les listes, on peut accéder aux flots en utilisant les méthodes *isEmpty*, *head* et *tail*. Par contre, le filtrage de motif n'est pas disponible pour les flots.

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

```
def print(xs: Stream[a]): unit =
  if (xs.isEmpty) () else { Console.println(xs.head); print(xs.tail) }
```

En fait, les flots supportent presque toutes les méthodes définies sur les listes.

Example: Pour trouver le 2ème nombre premier entre *1000* et *10000* :

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


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 *Stream.cons*. A.d. à la place de *x* :: *xs* on écrit *Stream.cons*(*x*, *xs*).

L'opérateur infixe ::: est remplacé par l'opérateur *append*. A.d. à la place de *xs* ::: *ys* on écrit *xs append ys*. *append* est défini ainsi :

```
def append[b >: a](rest: ⇒ Stream[b]): Stream[b];
  if (isEmpty) rest
  else Stream.cons(head, tail.append(rest));
```

- Notez que *append* prend en argument un flot dont les éléments peuvent être d'un supertype des éléments du récepteur.
- Cela s'exprime par la borne inférieure $>: a$ de la variable de type b.
- Le résultat est dans tous les cas un flot dont les éléments sont du type maximum des éléments des opérandes.

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.

```
trait Stream[+a] extends Seq[a] {
  def isEmpty: boolean;
  def head: a;
  def tail: Stream[a];
  ...
}
```

Les autres méthodes de *Stream* sont définies en termes de ces 3 méthodes. Elles sont toutes modélisées d'après les méthodes de la classe *List* :

```
def length: int;
def append[b >: a](rest: => Stream[b]): Stream[b];
def elements: Iterator[a];
def init: Stream[a];
def last: a;
def take(n: int): Stream[a];
```



```
def drop(n: int): Stream[a];
def apply(n: int): a;
def at(n: int): a;
def takeWhile(p: a => Boolean): Stream[a];
def dropWhile(p: a => Boolean): Stream[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 foldLeft[b](z: b)(f: (b, a) => b): b;
def foldRight[b](z: b)(f: (a, b) => b): b;
def reduceLeft[b >: a](f: (b, b) => b): b;
def reduceRight[b >: a](f: (b, b) => b): b;
def flatMap[b](f: a => Stream[b]): Stream[b];
def reverse: Stream[a];
def copyToArray[b >: a](xs: Array[b], start: int): int;
def zip[b](that: Stream[b]): Stream[Pair[a, b]];
def print: unit;
def toString(): String;
```

Flots concrets

Il y a aussi deux implantations concrètes de flots qui sont produites par *Stream.empty* et *Stream.cons*.

On peut les définir comme suit (voir plus bas pour une optimisation).

```
object Stream {
  val empty: Stream[All] = new Stream[All] {
    def isEmpty = true;
    def head: All = error("head of empty stream");
    def tail: Stream[All] = error("tail of empty stream");
  }
  def cons[a](hd: a, def tl: Stream[a]) = new Stream[a] {
    def isEmpty = false;
    def head: a = hd;
    def tail: Stream[a] = tl;
  }...
```


- La seule différence importante entre les implantations de *List* et de *Stream* concerne le statut de *tl*, le deuxième paramètre de *Stream.cons*.
- Ce paramètre est précédé de **def**; c'est pourquoi l'argument correspondant ne sera pas évalué lors de son passage à *Stream.cons*.
- Au contraire, un tel argument sera évalué à chaque fois que *tail* sera appelée sur l'objet flot.

Évaluation paresseuse

- L'implantation présentée a un problème d'efficacité potentiel : si *tail* est appelée plusieurs fois sur un objet *cons* la queue du flot sera évaluée plusieurs fois.
- Ce problème peut être évité en stockant le résultat de la première évaluation de *tail* et en réutilisant le résultat stocké à chaque nouvel appel à *tail*.

Exercise: Écrire une nouvelle version de *Stream.cons* qui implante cette amélioration.

L'implantation des flots en action

Pour vérifier que l'implantation des flots se comporte réellement efficacement, observons la trace d'exécution de l'exemple consistant à calculer le 2ème nombre premier entre *1000* et *10000*.

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

Stream.range(*1000*, *10000*).*filter*(*isPrime*).*at*(*1*)

→ *par expansion de* *Stream.range*

(**if** (*1000* ≥ *10000*) *Stream.empty*
else *Stream.cons*(*1000*, *Stream.range*(*1000* + *1*, *10000*)))
.*filter*(*isPrime*).*at*(*1*)

→ *par évaluation du* *if*

Stream.cons(*1000*, *Stream.range*(*1000* + *1*, *10000*)).*filter*(*isPrime*).*at*(*1*)

→ ...

...

→ *par expansion de filter,*
en notant $C_1 = Stream.cons(1000,\ Stream.range(1000 + 1,\ 10000))$
$(\textbf{if}\ (C_1.isEmpty)\ \textbf{this}$
$\textbf{else if}\ (isPrime\,(C_1.head))\ Stream.cons\,(C_1.head,\ C_1.tail.filter\,(isPrime))$
$\textbf{else}\ C_1.tail.filter\,(isPrime))\,.at\,(1)$

→ *par évaluation du if*

$(\textbf{if}\ (isPrime\,(C_1.head))\ Stream.cons\,(C_1.head,\ C_1.tail.filter\,(isPrime))$
$\textbf{else}\ C_1.tail.filter\,(isPrime))\,.at\,(1)$

→ *par évaluation de head*

$(\textbf{if}\ (isPrime\,(1000))\ Stream.cons\,(C_1.head,\ C_1.tail.filter\,(isPrime))$
$\textbf{else}\ C_1.tail.filter\,(isPrime))\,.at\,(1)$

→ *par évaluation de isPrime*

$(\textbf{if}\ (\textbf{false})\ Stream.cons\,(C_1.head,\ C_1.tail.filter\,(isPrime))$
$\textbf{else}\ C_1.tail.filter\,(isPrime))\,.at\,(1);$

→ *par évaluation du if*

$C_1.tail.filter\,(isPrime)\,.at\,(1);$

...

→ *par évaluation de tail*

$Stream.range\,(1001,\ 10000)\,.filter\,(isPrime)\,.at\,(1)$

Le processus continue ainsi jusqu'à atteindre

$Stream.range\,(1009,\ 10000)\,.filter\,(isPrime)\,.at\,(1)$

→ *par évaluation de filter,*

en notant $C_2 = Stream.cons(1009,\ Stream.range(1009 + 1,\ 10000))$
$Stream.cons\,(1009,\ C_2.tail.filter\,(isPrime))\,.at\,(1)$

→ *par évaluation de at*

$C_2.tail.filter\,(isPrime))\,.at\,(0)$

→ *par évaluation de at, expansion de* C_2

$Stream.range\,(1010,\ 10000)\,.filter\,(isPrime)\,.at\,(0)$

Le processus continue encore jusqu'à :

Stream.range(*1013*, *10000*).*filter*(*isPrime*).*at*(*0*)

→ *par évaluation de filter,*

en notant C_3 = *Stream.cons*(*1013*, *Stream.range*(*1013* + *1*, *10000*))
Stream.cons(*1013*, C_3.*tail.filter*(*isPrime*)).*at*(*0*)

→ *par évaluation de at*

1013

On constate que seule la partie des éléments de la liste nécessaire au calcul a été créée.

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 également 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] = Stream.cons(n, from(n+1));
val integers = from(0);
```

En partant des 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);
```

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.
- Éliminer de la liste tous les multiples de *2*.
- Le premier élément de la liste restante est *3*, qui est premier.
- Éliminer 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 modélisable par la fonction suivante.

```
def sieve(s: Stream[int]): Stream[int] =
  Stream.cons(s.head, sieve(s.tail filter (x => x % s.head != 0)));
val primes = sieve(from(2));
```

Maintenant pour voir la liste de tous les nombres premiers, afficher simplement la liste : *primes.print*

Itérer avec les flots

Les flots, avec leur évaluation retardée, sont un puissant outil de modélisation, car ils nous permettent de composer les modules d'un système d'une manière impossible à reproduire avec des variables d'état.

Par exemple, il est possible de s'intéresser à une série temporelle dans son ensemble, plutôt qu'à la valeur de la variable à un instant précis.

En particulier, des itérations complètes peuvent être transformées en objets qui peuvent eux-même être manipulés.

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

Cette fonction calculait la racine carrée en appliquant de manière itérative la fonction :

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

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

```
def sqrtStream(x: double): Stream[double] = {
  val guesses = Stream.cons(1, guesses map (guess ⇒ improve(guess, x)));
  guesses
}
```

On a alors :

```
> sqrtStream print
1
1.5
1.4166666666665
...
```


Un autre exemple consiste à calculer 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 considérer le flot des éléments à sommer de la série.

```
def piSummands(n: int): Stream[double] =
  Stream.cons(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 :

```
> piStream print
4.0, 2.666666666666667, 3.466666666666667, 2.8952380952380956,
3.33968253968254, 2.976046176046176, 3.283738483738484,
3.017071817071817, ...
```

Exercise: La fonction *partialSums* calcule pour un flot donné *xs* un nouveau flot $s_0, s_1, s_2, \ldots,$ où chaque s_i est donné par :

$$s_i \quad = \quad \sum_{j=0}^{i} xs_j$$

Complétez l'implantation suivante de *partialSums*.

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


Transformations de 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 suites* qui convertit une suite d'approximations en une autre qui converge plus rapidement.
- Un tel accélérateur, dû à Leonhard Euler, un mathématicien suisse du 18ème 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

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

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(0);
  val s1 = s(1);
  val s2 = s(2);
  Stream.cons(
    s2 - (s2 - s1) * (s2 - s1) / (s0 - 2 * s1 + s2),
    eulerTransform(s.tail));
}
```

On peut faire la démonstration de l'accélération d'Euler pour notre suite d'approximations de *pi*.

```
> eulerTransform(piStream).print
3.166666666666667, 3.1333333333333337, 3.1452380952380956,
3.1396825396825396, 3.142712842712843, 3.140881340881341,
3.142071817071817, 3.1412548236077646, ...
```


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 *tableau*), dans lequel chaque flot est le transformé du précédent :

```
def tableau(transform: Stream[double] ⇒ Stream[double],
            s: Stream[double]): Stream[Stream[double]] =
  Stream.cons(s, tableau(transform, transform(s)));
```

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

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

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 voir à l'œuvre cette sorte de "super-accélération" de la suite de π :

```
> accelerate(eulerTransform, piStream)
4.0, 3.166666666666667, 3.142105263157895, 3.1415993573190044,
3.141592714033778, 3.1415926539752923, 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.

Itérateurs

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

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 grâce à une fonction *next*.

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

```
trait Iterator[+a] {
  def hasNext: boolean;
  def next: a;
}
```

hasNext retourne ***true*** ssi l'itérateur a encore des éléments.

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

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 existe les méthodes *length*, *append*, *take* et *drop*, *map* et *flatMap*, *foreach*, *foldLeft* et *foldRight*, *zip*, etc.

Sous-classes de Iterator

Voici deux sous-classes de *Iterator*.

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

```
final class EmptyIterator[c] extends Iterator[c] {
  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] {
  var cur: List[a] = xs;
  def hasNext: boolean = cur.nonEmpty;
  def next: a = { val x = cur.head; cur = cur.tail; x }
}
```

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

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

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


Utilisation des itérateurs

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

Example: 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;
  Stream.cons(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 parcouru plus d'une fois (car la première utilisation va en fait le modifier !)

En particulier, les techniques d'accélération de séries telles que la transformée d'Euler ne peuvent pas être utilisées telles quelles, car elles accèdent à une suite plusieurs fois.

Implantation des méthodes d'Iterator

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

Ici, *Iterator.**this*** indique l'identité de l'objet *Iterator* englobant (plutôt que l'objet créé par le template lié au ***new***).

```
trait Iterator[+a] {
  def append[b >: a](that: Iterator[b]) = new Iterator[b] {
    def hasNext = Iterator.this.hasNext || that.hasNext;
    def next = if (Iterator.this.hasNext) Iterator.this.next else that.next;
  }

  def map[b](f: a ⇒ b) = new Iterator[b] {
    def hasNext: boolean = Iterator.this.hasNext;
    def next: b = f(Iterator.this.next);
  }
```

Exercise: Implanter la méthode

def *foreach*(*f*: *a* ⇒ *unit*): *unit*;

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

Exercise: Implanter une méthode

def *dropWhile*(*p*: *a* ⇒ *boolean*): *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*.

Itérateurs ou flots: que choisir ?

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 ?

On pourrait dire que les itérateurs sont les flots "du pauvre".

Leurs avantages sont :

- Ils peuvent être plus rapides, vu qu'aucun objet *cons* n'est construit.
- Ils peuvent être implantés même dans des langages primaires, sans paramètres avec appel par nom, et sans fonctions d'ordre supérieur.

Leurs inconvénients sont :

- Une dépendance d'ordre : l'ordre des éléments dans l'itérateur doit correspondre à l'ordre dans lequel on veut appeler l'opération *next*.
- Pas de réutilisation : une fois qu'un itérateur a avancé, l'ancien état est perdu ; on ne peut donc pas parcourir un itérateur plus d'une fois.