Programmation Avancée (INF441)

Cascades, suspensions et flots

François Pottier

24 mai 2016

Précédemment...

Nous avons posé une question :

- Comment séparer modulairement producteur et consommateur ?

Nous avons aperçu deux familles de solutions :

- les « réducteurs » ou « fold », où le producteur contrôle ;
- les itérateurs, où le consommateur contrôle.

Précédemment...

Les réducteurs peuvent être plus faciles à écrire que les itérateurs.

- ils s'écrivent naturellement en style itératif ou récursif, si besoin
- ils s'interrompent facilement pour appeler le consommateur

Les itérateurs peuvent être plus faciles à utiliser que les réducteurs.

- le consommateur choisit quand et à qui demander un élément

Aujourd'hui

Les itérateurs sont-ils difficiles à écrire ?

Cela peut être vrai si l'on s'y prend mal, par exemple

- si l'on choisit des itérateurs modifiables (plus complexes), et
- si l'on n'utilise pas les clôtures (qui facilitent la suspension),
- ce qui est le style encouragé (voire imposé) en Java...

Aujourd'hui

Les itérateurs sont-ils difficiles à écrire ?

Pas nécessairement.

Je vais en donner une vision simple aujourd'hui,

- d'abord en revenant sur les itérateurs immuables, ou cascades ;
- puis en introduisant une variante, les flots.

Itérateur (modifiable)

Un itérateur produit un élément à la demande.

```
type 'a iterator =
  unit -> 'a option
```

C'est un objet, un service.

Il est nécessairement éphémère, utilisable une seule fois.

C'est le style encouragé en Java, via l'interface `Iterator`.

Itérateur (modifiable)

Un itérateur peut sembler pénible à écrire. (Poly, exercices 4.9 et 4.10.)

```
let interval (j : int) (k : int) : int iterator =
  let next = ref j in
  fun () ->
    let c = !next in
    if c < k then begin
      next := c + 1; Some c
    end
    else None
```

Il faut maintenir un état interne modifiable et écrire le code de façon à s'interrompre à chaque élément produit puis reprendre au prochain appel.

Le code ne semble pas très naturel.

Vers un style différent

Toutefois, ce n'est pas une fatalité.

Il existe un style qui rend les itérateurs faciles à écrire.

- ce style est à base de clôtures,
- car elles facilitent l'interruption et le redémarrage ;
- et il n'est pas compliqué !

Dans la suite, je considère d'abord et surtout des itérateurs immuables, ou cascades, mais les itérateurs modifiables s'en déduisent.

Cascade

Une cascade produit à la demande un élément et une cascade qui représente le reste des éléments.

```
type 'a cascade_now =       (* a cascade whose head *)
| Nil                       (*     is available now *)
| Cons of 'a * 'a cascade

and 'a cascade =            (* a cascade whose head *)
  unit -> 'a cascade_now    (*     must be demanded *)
```

Une cascade est une liste dont chaque élément est calculé à la demande.

Cascade

Nous avons déjà rencontré ce type, sous le nom `'a immutable_iterator`.

J'appelle maintenant cela cascade car je trouve ce mot court et imagé.

La définition du type `'a cascade` ne dit pas si une cascade est :

- éphémère, utilisable une seule fois ; ou
- persistante, utilisable autant de fois qu'on le souhaite.

Les deux ont un sens. Dans la suite, je construis uniquement des cascades persistantes. Elles sont plus simples, car sans état modifiable.

Une cascade représente une suite d'éléments.

Cascades pour tous

L'idée est simple : les cascades sont (presque) des listes.

Elles sont (presque) aussi faciles d'emploi que les listes.

Faciles à produire, à consommer, à transformer...

Constructeurs

Donnons-nous deux constructeurs :

```
let nil           = Nil
let cons who x xs = Cons (x, xs)
```

Dans le code en ligne, la fonction `cons` affiche un message, ce qui permet d'observer à quel moment et dans quel ordre les calculs ont lieu.

Produire une cascade

Voici la cascade des entiers de `j` inclus à `k` exclus :

```
let rec interval (j : int) (k : int) : int cascade =
  fun () ->   (* delay the production of the head *)
    if j < k then
      cons "interval" j (interval (j + 1) k)
    else
      nil
```

Un appel à `interval j k` consomme un temps $O(1)$.

Il alloue une clôture, et c'est tout. L'exécution du corps est retardée.

Cette clôture, lorsqu'on l'appelle, consomme un temps $O(1)$.

Elle alloue soit un bloc `Cons` et une clôture, soit un bloc `Nil`.

Produire une cascade

Voici la cascade infinie des entiers à partir de `j` :

```
let rec from (j : int) : int cascade =
  fun () ->
    cons "from" j (from (j + 1))
```

Contrairement à une liste, une cascade peut être infinie.

L'appel récursif `from (j + 1)` est retardé.

Il n'a lieu que lorsque le client exige le premier élément, à savoir `j`.

- une fonction récursive qui termine en temps constant !

Produire une liste infinie ?

Si on tente de produire une liste infinie dans le même style :

```
let rec from (j : int) : int list =
  j :: from (j + 1)
```

L'appel récursif `from (j + 1)` n'est pas retardé.

Du coup (dans une session interactive) :

```
# from 0;;
Stack overflow during evaluation (looping recursion?).
```

Consommer une cascade

La production des éléments est exigée par le consommateur :

```
let rec find (p : 'a -> bool) (xs : 'a cascade) : 'a option =
  match xs() with      (* demand the production of the head *)
  | Nil ->
      None
  | Cons (x, xs) ->
      if p x then
        Some x
      else
        find p xs
```

Le code s'écrit comme si `xs` était une liste, sauf qu'il faut écrire `xs()` pour exiger le calcul de la tête.

Transformer une cascade

Un « transformateur » est à la fois producteur et consommateur :

```
let rec map (f : 'a -> 'b) (xs : 'a cascade) : 'b cascade =
  fun () ->          (*  delay *)
    match xs() with (* demand *)
    | Nil ->
        Nil
    | Cons (x, xs) ->
        cons "map" (f x) (map f xs)
```

Un appel `let ys = map f xs in ...` termine en temps $O(1)$.

Si un jour un consommateur appelle `ys()`, alors et alors seulement :

- `xs()` sera appelé pour obtenir un élément `x`,
- puis on calculera `f x`,
- qui sera renvoyé en tant que premier élément de la cascade `ys`.

Pipeline

On peut composer producteur, transformateurs, consommateur :

```
let m : int option =
  find (fun x -> x mod 7 = 0) (map (fun x -> 2 * x) (from 33))
```

On obtient un pipeline.

Pipeline

La bibliothèque d'OCaml définit un opérateur d'application de fonction où l'argument est à gauche. Il est associatif à gauche.

```
let (|>) x f = f x
```

Le pipeline s'écrit alors plus lisiblement :

```
let m : int option =
  from 33
    |> map (fun x -> 2 * x)
    |> find (fun x -> x mod 7 = 0)
```

On pourrait facilement construire des pipelines plus complexes.

Évaluation à la demande

L'évaluation se fait à la demande. Dans une session interactive :

```
# let m : int option =
    from 33
      |> map (fun x -> 2 * x)
      |> find (fun x -> x mod 7 = 0);;
from: producing 33
map: producing 66
from: producing 34
map: producing 68
from: producing 35
map: producing 70
val m : int option = Some 70
```

Lorsque `find` exige un élément de `map`, `map` exige un élément de `from`.

`from` produit 33 et le renvoie à `map`, qui calcule 66 et le renvoie à `find`.

`find` n'est pas satisfait, donc exige l'élément suivant ; et ainsi de suite.

Transformer une cascade, avec état

Étant donnée une cascade `xs`, voici la cascade de ses sommes partielles :

```
let rec sum accu (xs : int cascade) : int cascade =
  fun () ->
    match xs() with
    | Nil ->
        Nil
    | Cons (x, xs) ->
        let accu = accu + x in
        cons "sum" accu (sum accu xs)
```

Il n'y a pas d'état modifiable. La cascade produite est persistante.

Deux producteurs pour un consommateur

Ici, on produit une suite de paires à partir de deux suites d'éléments :

```
let rec zip xs ys =
  fun () ->
    match xs(), ys() with
    | Nil, _
    | _, Nil ->
        Nil
    | Cons (x, xs), Cons (y, ys) ->
        Cons ((x, y), zip xs ys)
```

À chaque fois qu'un nouvel élément est exigé, `xs` et `ys` sont interrogés.

Évaluation en espace constant

L'espace utilisé reste $O(1)$, même si on itère sur une longue cascade :

```
let m : int option =
  from 0
    |> sum 0
    |> find (fun s -> s >= 1000000000)          (* one billion *)
```

Le nombre total d'objets alloués dans le tas est $O(n)$ où n est le nombre d'éléments exigés par `find`.

Le nombre d'objets vivants (accessibles) à un instant donné reste $O(1)$.

Intérêt des cascades

En résumé, nous avons :

- modularité : séparation entre producteurs, transformateurs, consommateurs ; facilité à assembler des pipelines
- abstraction : les transferts de contrôle sont complexes, mais (souvent) on peut ne pas y penser
- efficacité : calcul en espace O(1) possible, même si les séquences sont longues, voire infinies
- évaluation à la demande : on ne calcule que ce qui est exigé

Le prix à payer est un facteur constant en temps, car on alloue beaucoup.

Peut-on employer ce style pour obtenir un itérateur ?

D'une cascade à un itérateur modifiable

Oui. On peut adapter une cascade pour obtenir un itérateur modifiable :

```
let cascade_to_iterator (xs: 'a cascade): unit -> 'a option =
  let current = ref xs in
  fun () ->
    match (!current)() with
    | Nil ->
        None
    | Cons (x, xs) ->
        current := xs;
        Some x
```

On peut écrire cet adaptateur une fois pour toutes.

La conversion en sens inverse est possible également mais produirait une cascade non persistante. Les flots résoudront ce problème.

Peut-on employer ce style lorsque le producteur est récursif, par exemple pour itérer sur un arbre ?

Produire les éléments d'un arbre

Soit un type d'arbres binaires où chaque nœud porte un élément :

```
type 'a tree =
| Leaf
| Node of 'a tree * 'a * 'a tree
```

Produire les éléments d'un arbre

Vous savez construire la liste des éléments d'un arbre dans l'ordre infixe :

```
let rec elements (t : 'a tree) (accu : 'a list) : 'a list =
  match t with
  | Leaf ->
      accu
  | Node (t0, x, t1) ->
      elements t0 (x :: elements t1 accu)
```

`elements t accu` produit la liste des éléments de `t` suivie de la liste `accu`.

Cette formulation évite la concaténation `@` et coûte un temps $O(n)$.

Le code est récursif et utilise un espace $O(h)$ sur la pile implicite.

Produire les éléments d'un arbre

On construit de même la cascade des éléments d'un arbre :

```
let rec elements t (accu : 'a cascade_now) : 'a cascade =
  fun () -> elements_now t accu

and elements_now t (accu : 'a cascade_now) : 'a cascade_now =
  match t with
  | Leaf ->
      accu
  | Node (t0, x, t1) ->
      elements_now t0 (Cons (x, elements t1 accu))
```

Le code est récursif mais utilise un espace $O(1)$ sur la pile implicite.

On peut lire `elements_now` comme une boucle qui descend le long de la branche de gauche tout en empilant sur `accu` les sous-arbres droits.

`accu` occupe un espace $O(h)$ dans le tas.

Duplication de calculs

La définition de `from33double` ne provoque aucun calcul immédiat :

```
let from33double =
  from 33 |> map (fun x -> 2 * x)
```

Si on l'utilise deux fois, les éléments 66, 68, 70 sont calculés deux fois :

```
let m : int option =
  from33double |> find (fun x -> x mod 7 = 0)

let m : int option =
  from33double |> find (fun x -> x mod 13 = 0)
```

Duplication de calculs

D'où problème si un producteur est branché sur deux consommateurs :

```
let m : (int * int) option =
  let xs = from 0 in                    (* one producer *)
  zip xs (xs |> sum 0)                  (* used twice *)
    |> find (fun (x, s) -> s >= 50)
```

On combine la cascade des entiers 0, 1, 2, ... et la cascade de leurs sommes partielles 0, 1, 3, ... et on s'arrête dès que la somme atteint 50.

Facile à écrire. Joli. Mais chaque élément de `from 0` est calculé deux fois.

Éviter la duplication

Comment faire en sorte que chaque élément soit calculé au plus une fois ?

Pour cela, nous devons

- non seulement retarder le calcul de chaque élément,
- mais aussi mémoiser son résultat.

Nous éviterons ainsi d'effectuer deux fois un même calcul.

Calculs retardés

Jusqu'ici, nous avons utilisé une fonction de type `unit -> 'a` pour décrire un calcul en attente dont le résultat pourra être exigé plus tard.

Nous avons employé deux opérations élémentaires, retarder et exiger :

```
fun () -> ...  (* delay *)

xs()           (* demand, or force *)
```

Calculs retardés

On peut imaginer un type abstrait des calculs en attente.

Sa signature est :

```
type 'a delayed
val delay: (unit -> 'a) -> 'a delayed
val force: 'a delayed -> 'a
```

et son implémentation (triviale) :

```
type 'a delayed = unit -> 'a
let delay k = k
let force k = k()
```

Calculs retardés et mémoisés

On peut imaginer une autre implémentation de la même signature :

```
type 'a delayed
val delay: (unit -> 'a) -> 'a delayed
val force: 'a delayed -> 'a
```

avec la propriété que si `force d` est appelé deux fois, alors le calcul décrit par `d` n'est effectué qu'une fois.

Voyez-vous comment réaliser cela ?

Implémentation approximative

On n'a même pas besoin de modifier `delayed` ni `force` :

```
type 'a delayed = unit -> 'a
let force k = k()
```

Le tout est de prévoir la mémoisation de chaque calcul suspendu :

```
let delay (f : unit -> 'a) : unit -> 'a =
  let state = ref None in
  fun () ->
    match !state with
    | Some a -> a          (* result already available *)
    | None   ->            (* result     not available *)
        let a = f() in     (* compute it,              *)
        state := Some a;   (* store it for next time,  *)
        a                  (* and return it            *)
```

Implémentation fausse à cause de la ré-entrance, des exceptions, et de la concurrence. (Poly, exercices 1.11 et 1.12.) Mais l'idée y est.

Suspensions

Le module `Lazy` d'OCaml propose des suspensions primitives :

```
type 'a t
val from_fun: (unit -> 'a) -> 'a t (* delay *)
val force: 'a t -> 'a              (* force *)
```

Au lieu de `from_fun (fun () -> ...)` on peut aussi écrire `lazy (...)`.

Ainsi (dans une session interactive) :

```
# let s = lazy (print_endline "Coucou!"; 3);;
val s : int Lazy.t = <lazy>
# Lazy.force s;;
Coucou!
- : int = 3
# Lazy.force s;;
- : int = 3
```

On voit que le calcul est retardé et mémoisé.

Des cascades aux flots

Modifions le type des cascades pour que chaque élément soit non seulement calculé à la demande mais aussi mémoisé.

```
type 'a stream_now =        (* a stream whose head *)
| Nil                       (*    is available now *)
| Cons of 'a * 'a stream

and 'a stream =             (* a stream whose head *)
  'a stream_now Lazy.t      (*    must be demanded *)
```

On appelle ceci un flot ou stream.

Les flots jouent un rôle central en Haskell.

Des cascades aux flots

Les fonctions qui produisent ou consomment des flots s'écrivent comme celles qui produisent ou consomment des cascades (*).

Les deux opérations élémentaires, retarder et exiger, sont maintenant :

```
lazy (...)     (* delay *)

Lazy.force xs (* force *)
```

(*) On pourrait faire en sorte que cascades et flots aient le même type, et éviter ainsi de dupliquer ces fonctions. Je n'approfondis pas cette idée.

Produire un flot

```
let rec interval (j : int) (k : int) : int stream =
  lazy (                                    (* delay *)
    if j < k then
      cons "interval" j (interval (j + 1) k)
    else
      nil
  )
```

Consommer un flot

```
let rec find (p : 'a -> bool) (xs : 'a stream) : 'a option =
  match Lazy.force xs with                          (* demand *)
  | Nil ->
      None
  | Cons (x, xs) ->
      if p x then
        Some x
      else
        find p xs
```

Transformer un flot

```
let rec map (f : 'a -> 'b) (xs : 'a stream) : 'b stream =
  lazy (                                          (*  delay *)
    match Lazy.force xs with                      (* demand *)
    | Nil ->
        Nil
    | Cons (x, xs) ->
        cons "map" (f x) (map f xs)
  )
```

Mémoisation des calculs

Comme dans le cas des cascades, la définition de `from33double` ne provoque aucun calcul immédiat (ici, dans une session interactive) :

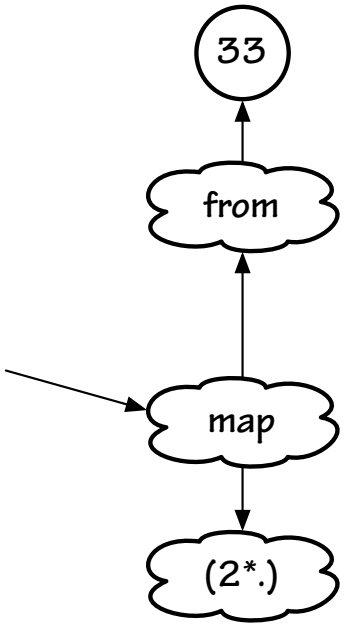
```
# let from33double =
    from 33
      |> map (fun x -> 2 * x);;
val from33double : int stream = <lazy>
```

À cet instant, `from33double` est représenté par deux suspensions non évaluées, rien d'autre.

L'une, produite par `map`, pointe vers l'autre, produite par `from`.

Mémoisation des calculs

Un nuage représente une suspension ou une clôture.



Une suspension créée par `from` a une variable libre, `j`.

Une suspension créée par `map` a deux variables libres, `f` et `xs`.

Mémoisation des calculs

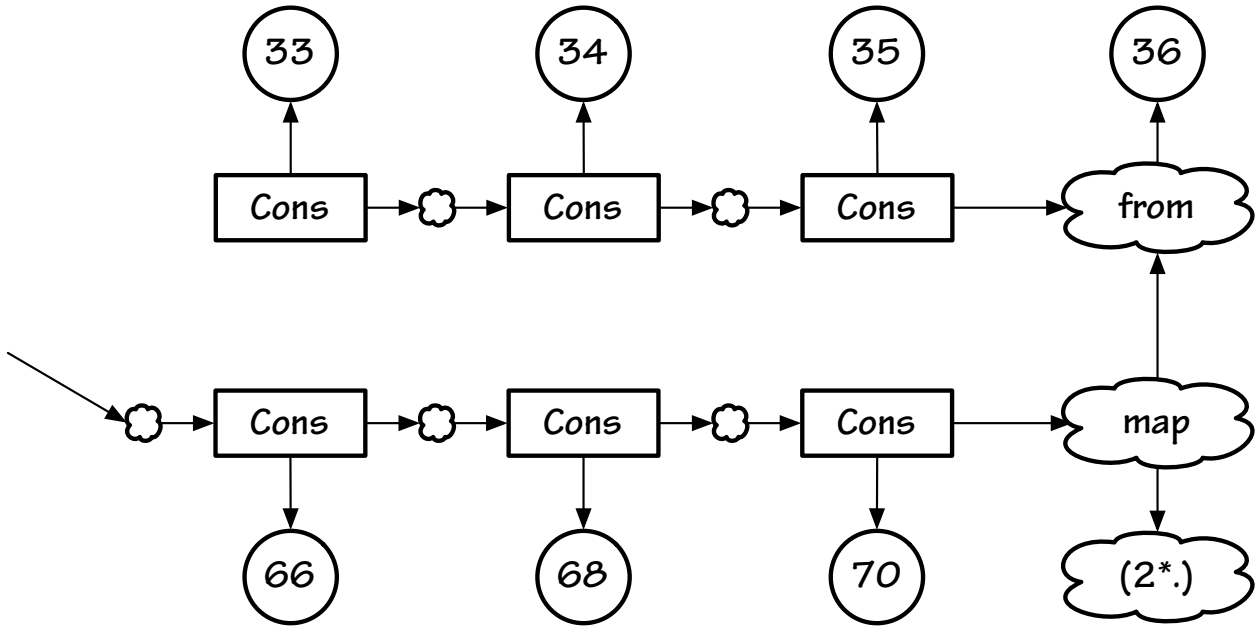
Comme dans le cas des cascades, si on utilise une première fois ce flot, une partie de ses éléments sont calculés :

```
# let m : int option =
    from33double |>
      find (fun x -> x mod 7 = 0);;
from: producing 33
map: producing 66
from: producing 34
map: producing 68
from: producing 35
map: producing 70
- : int option = Some 70
```

À cet instant, `from33double` est représenté par trois blocs `Cons` et trois suspensions évaluées, chaînés, et deux suspensions non évaluées...

Mémoisation des calculs

Un mini-nuage représente une suspension évaluée.



Notez que les trois cellules `Cons` situées en haut sont inaccessibles.

Mémoisation des calculs

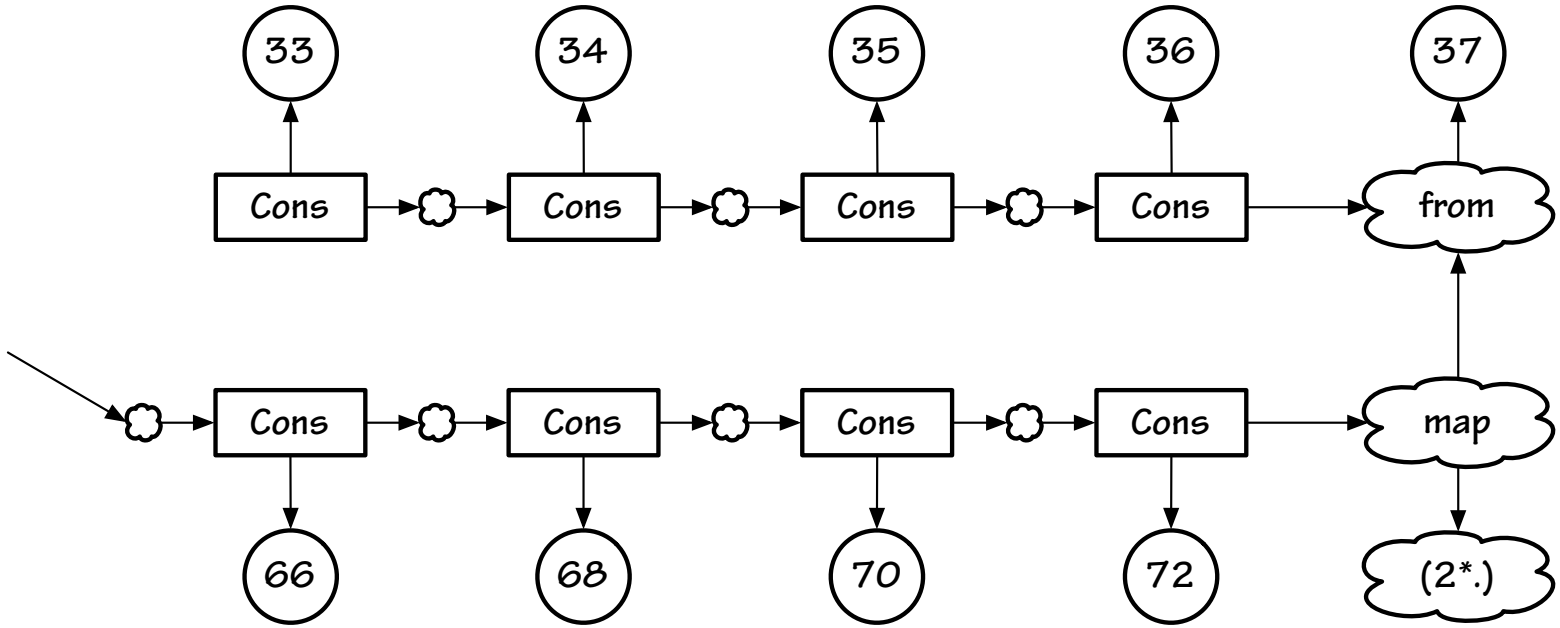
À la différence du cas des cascades, si on utilise une deuxième fois ce flot, les éléments déjà calculés ne sont pas re-calculés :

```
# let m : int option =
    from33double |>
      find (fun x -> x mod 12 = 0);;
from: producing 36
map: producing 72
val m : int option = Some 72
```

À cet instant, `from33double` est représenté par quatre blocs `Cons` et quatre suspensions évaluées, chaînés, et deux suspensions non évaluées...

Mémoisation des calculs

Les quatre cellules `Cons` situées en haut sont inaccessibles.



Les quatre cellules `Cons` situées en bas restent accessibles tant qu'existe la variable `from33double`, qui contient l'adresse de ce flot.

Duplication de calculs

Si un producteur `xs` est branché sur deux consommateurs, chaque élément de `xs` est exigé deux fois, mais n'est calculé qu'une fois :

```
let m : (int * int) option =
  let xs = from 0 in                    (* one producer *)
  zip xs (xs |> sum 0)                  (* used twice *)
    |> find (fun (x, s) -> s >= 50)
```

L'espace utilisé est toujours $O(1)$, car au fur et à mesure que `find` avance dans le flot `zip ...`, les éléments précédents deviennent inaccessibles.

Toutefois, attention, car si on écrivait ceci :

```
let xs = from 0
let m : (int * int) option =
  zip xs (xs |> sum 0)
    |> find (fun (x, s) -> s >= 50)
```

alors les éléments du flot `xs` resteraient accessibles et l'espace utilisé serait plus important.

Mémoisation des calculs

Parce qu'une suspension est un objet modifiable, un flot est lui aussi un objet modifiable.

Sa représentation en mémoire change au cours du temps,

- un flot non évalué peut occuper un espace $O(1)$ ~ itérateur
- un flot entièrement évalué occupe un espace $O(n)$ ~ liste
- un flot peut aussi être partiellement évalué

On peut considérer les flots comme un hybride entre itérateurs et listes.

Est-ce une bonne idée ?

Les flots ont un intérêt :

- l'utilisateur ne sait pas si le flot est déjà évalué ou non

Il s'utilise de la même manière dans les deux cas ; c'est bien.

Les flots ont un inconvénient :

- l'utilisateur ne sait pas si le flot est déjà évalué ou non

On ne sait pas quel espace il occupe ; c'est mal.

On ne sait pas combien de temps il faudra pour obtenir chaque élément.

On peut le savoir, si on réfléchit, mais ce n'est pas évident a priori.

D'un itérateur à un flot

On adapte facilement un itérateur modifiable pour obtenir un flot :

```
let rec iterator_to_stream
  (it : unit -> 'a option) : 'a stream =
  lazy (
    match it() with
    | None ->
        Nil
    | Some x ->
        Cons (x, iterator_to_stream it)
  )
```

Grâce à la mémoisation, ce flot est persistant.

Si on construisait une cascade ainsi, elle serait éphémère.

Danger : une fois `it` ainsi transformé en flot, il ne faut plus l'utiliser excepté (implicitement) en consommant le flot.

D'un itérateur à un flot

Voici par exemple un itérateur modifiable qui lit un fichier :

```
let read filename : unit -> char option =
  let channel = open_in filename in
  fun () ->
    try
      Some (input_char channel)
    with End_of_file ->
      close_in channel;
      None
```

On le transforme facilement en flot. Le fichier sera ainsi lu à la demande.

Danger : le fichier ne sera fermé que si le flot est évalué jusqu'au bout.

Tri par fusion paresseux

On peut écrire `merge` et `sort` sur les flots comme sur les listes.

(Voir le code en ligne.)

Grâce à la mémoisation :

- la liste argument n'est construite qu'une fois
  - bien que `sort` la parcoure deux fois
- la liste triée n'est construite qu'une fois
  - même si elle est utilisée ensuite plusieurs fois

Tri par fusion paresseux

Grâce au calcul à la demande,

- le premier élément est produit en temps $O(n)$
- et chaque élément suivant en temps $O(\log n)$

Le code est simple mais son comportement dynamique très subtil !

Conclusion / cascades et flots

Les cascades et les flots sont (souvent) faciles à écrire et faciles à utiliser.

Techniquement, c'est un style où :

- le consommateur contrôle quand les éléments sont produits, mais
- le producteur s'écrit naturellement, comme s'il construisait une liste.

Il peut être difficile de comprendre quand les calculs ont lieu, combien de temps ils demandent et combien d'espace occupent les calculs en attente.

Les « langages synchrones » (Esterel, Lustre, Lucid Synchrone, . . .) sont conçus pour mieux contrôler ces questions.

Rappels

- TD aujourd'hui : itérateurs en Java.
- TD la semaine prochaine : flots en OCaml.
- Date limite pour rendre votre projet : **31 mai 2016** à 23h59.
- Contrôle classant le **mardi 7 juin 2016**.