Programmation Avancée (INF441)

Réducteurs versus cascades

François Pottier

31 mai 2016

En bref

Aujourd'hui, nous étudions les « réducteurs », ou producteurs qui contrôlent un consommateur...

...et les comparons aux itérateurs / cascades, ou producteurs contrôlés par un consommateur.

- Peut-on « adapter » une cascade pour en faire un réducteur ?
  – oui, sans même avoir accès au code
- Peut-on « adapter » un réducteur pour en faire une cascade ?
  – non, sans accès au code
  – oui, si on peut accéder au code et le réécrire

Somme des éléments d'une liste

Nous avons rencontré une manière d'additionner les éléments d'une liste :

```
let rec sum xs =
  match xs with
  | []      -> 0
  | x :: xs -> x + sum xs
```

Ce code n'est pas le plus efficace, car il exige un espace $O(n)$, mais est le plus naturel au sens où il « remplace » simplement `[]` par `0` et `::` par `+`.

Somme des éléments d'une liste

Si on pense ainsi en termes de « remplacement », on voit que l'appel :

```
sum (a :: b :: ... z :: [])
```

est équivalent à :

```
a + (b + (... (z + 0)...))
```

ou, en notation polonaise inverse :

```
0 z +
  ...
  b +
  a +
```

Produit, maximum des éléments d'une liste

On pourrait calculer de la même façon le produit des éléments d'une liste :

```
let rec product xs =
  match xs with
  | []      -> 1
  | x :: xs -> x * product xs
```

ou bien leur maximum :

```
let rec maximum xs =
  match xs with
  | []      -> min_int
  | x :: xs -> max x (maximum xs)
```

Réduction d'une liste

Généralisons ce motif à un opérateur `op` et à un élément de départ `base` :

```
let rec reduce op xs base =
  match xs with
  | []      -> base
  | x :: xs -> op x (reduce op xs base)
```

Les opérations précédentes deviennent des cas particuliers :

```
let sum xs = reduce (+) xs 0
let product xs = reduce ( * ) xs 1
let maximum xs = reduce max xs min_int
```

La fonction `reduce` « réduit » une liste à une valeur qui la résume.

Elle s'appelle `List.fold_right` dans la bibliothèque d'OCaml.

C'est un producteur paramétré par un consommateur.

Réduction d'une liste

À nouveau, on voit que l'appel :

```
reduce op (a :: b :: ... :: z :: []) base
```

est équivalent à :

```
op a (op b (... (op z base)...))
```

ou, en notation polonaise inverse :

```
base z swap op
     ...
     b swap op
     a swap op
```

Si `op` est associatif, on peut paralléliser ce calcul, mais c'est une autre histoire (MapReduce)...

Réduction d'une liste

Quel est le type de `reduce` ? On pense naturellement à celui-ci :

```
val reduce: ('a -> 'a -> 'a) -> 'a list -> 'a -> 'a
```

Mais si on laisse OCaml inférer son type, on obtient :

```
val reduce: ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

En effet, rien n'oblige éléments et « résumé » à avoir le même type.

Affichage d'une liste

Par exemple, pour convertir une liste d'entiers en chaîne de caractères :

```
let nil       = "[]"
let cons x s = Printf.sprintf "%d :: %s" x s
let print xs = reduce cons xs nil
```

On voit que `print (1 :: 2 :: 3 :: 4 :: [])`
est égal à `cons 1 (cons 2 (cons 3 (cons 4 nil)))`
c'est-à-dire `"1 :: 2 :: 3 :: 4 :: []"`.

Et en effet, dans une session interactive :

```
# print [1;2;3;4];;
- : string = "1 :: 2 :: 3 :: 4 :: []"
```

Fold droite-gauche, ou bottom-up

`reduce` calcule de droite à gauche et coûte $O(n)$ en espace.

On peut généraliser l'idée à n'importe quel type d'arbres et définir pour ces arbres une fonction « fold » qui calcule de bas en haut et coûte $O(h)$ en espace. (Poly, exercices 4.7 et 4.8.)

Fold gauche-droite

On peut préférer calculer de gauche à droite pour un coût $O(1)$ en espace.

On obtient une fonction `fold_left`, très proche de la fonction `iter` rencontrée la semaine dernière.

Partons de `iter` pour comprendre comment il faut écrire `fold_left`...

Définition de `iter`

Voici comment s'écrit `iter`, dans sa variante gauche-droite :

```
let rec iter consume xs =
  match xs with
  | [] ->
      ()
  | x :: xs ->
      consume x;
      iter consume xs
```

Voici son type :

```
val iter: ('a -> unit) -> 'a list -> unit
```

Utilisation de `iter`

Voici une somme calculée à l'aide de `iter` :

```
let sum xs =
  let accu = ref 0 in
  List.iter (fun x -> accu := !accu + x) xs;
  !accu
```

Le consommateur doit mémoriser son état à l'aide d'une référence `accu` (un objet modifiable, alloué dans le tas).

Définition de `fold_left`

`fold_left` imite `iter`, mais de plus, se charge de transporter l'état du consommateur, ou accumulateur, entre deux appels au consommateur.

```
let rec fold_left consume accu xs =
  match xs with
  | [] ->
      accu
  | x :: xs ->
      let accu = consume accu x in
      fold_left consume accu xs
```

Ici, on ne connaît pas le type de `accu`. On suppose seulement que `consume` attend un état et un élément et renvoie un état nouveau.

Utilisation de `fold_left`

Ainsi, le consommateur n'a plus besoin d'allouer une référence :

```
let sum xs =
  List.fold_left (fun accu x -> accu + x) 0 xs
```

ou tout simplement :

```
let sum xs =
  List.fold_left (+) 0 xs
```

Ici, l'accumulateur est un entier : la somme partielle.

Type de `fold_left`

`fold_left` généralise `iter`. On retrouve `iter` à partir de `fold_left` si l'on transporte un accumulateur trivial de type `unit` :

```
let iter consume xs =
  fold_left (fun () x -> consume x) () xs
```

Le type de `iter` est :

```
val iter: ('a -> unit) -> 'a list -> unit
```

Voici celui de `fold_left` :

```
val fold_left: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

`fold_left` et `fold_right` ont le même type, excepté l'ordre des paramètres, qui est purement question de convention.

Spécification de `fold_left`

On voit que l'appel :

```
fold_left op base (a :: b :: ... :: z :: [])
```

est équivalent à :

```
op (... (op (op base a) b) ...) z
```

ce qui est beaucoup plus clair en notation polonaise inverse :

```
base a op
     b op
     ...
     z op
```

Interlude

Formulation d'un parcours en profondeur d'abord comme un `iter`.

(Code en ligne)

(Pas dans le poly.)

iter en Java

En Java 8, l'interface `Iterable` définit une méthode `forEach` :

```
public interface Iterable<E> {
  Iterator<E> iterator ();
  default void forEach (Consumer<E> consumer) { ... }
}
public interface Consumer<E> {
  void accept (E e);
}
```

C'est l'analogue d'une fonction `iter`.

Je reviendrai plus loin sur la notion de méthode par défaut.

Réducteur versus itérateur

Le type d'un réducteur (sans accumulateur) est :

```
type 'a iter =
  ('a -> unit) -> unit
```

Le type d'un itérateur (modifiable) est :

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

On peut « adapter » un itérateur pour en faire une cascade, c'est-à-dire écrire une fonction de type `'a iterator -> 'a cascade`, et en sens inverse également. (Exercices !)

Peut-on convertir entre réducteur et itérateur ?

D'un itérateur à un réducteur (OCaml)

Peut-on écrire un « adaptateur » de type `'a iterator -> 'a iter` ?

D'un itérateur à un réducteur (OCaml)

Peut-on écrire un « adaptateur » de type `'a iterator -> 'a iter` ?

Bien sûr. C'est une boucle qui interroge l'itérateur jusqu'à épuisement.

```
let rec foreach (it : 'a iterator) : 'a iter =
  fun (consume : 'a -> unit) ->
    match it() with
    | None ->
        ()
    | Some x ->
        consume x;
        foreach it consume
```

D'un itérateur à un réducteur (Java)

La méthode `forEach` de l'interface `Iterable` joue exactement le même rôle « d'adaptateur ».

Rappelez-vous ce que j'ai présenté tout à l'heure :

```
public interface Iterable<E> {
  Iterator<E> iterator ();
  default void forEach (Consumer<E> consumer) { ... }
}
public interface Consumer<E> {
  void accept (E e);
}
```

Comment cette méthode est-elle implémentée ?

D'un itérateur à un réducteur (Java)

L'implémentation est triviale. C'est à nouveau une boucle.

```
public interface Iterable<E> {
  Iterator<E> iterator ();
  default void forEach (Consumer<E> consumer) {
    for (E e : this)
      consumer.accept(e);
  }
}
```

En Java 1.8, une interface `I` peut définir une méthode `m` par défaut.

Une classe `C` qui veut satisfaire `I` a alors le choix d'implémenter ou non `m`. Si elle ne l'implémente pas, elle hérite de l'implémentation par défaut.

Itérateur ou réducteur (Java)

En Java, on a donc deux façons d'énumérer les éléments d'une collection.

Soit via un itérateur :

```
for (E e : c) { ... }
```

Soit via un appel à `forEach` :

```
c.forEach(e -> { ... });
```

Cela au revient au même si la collection utilise l'implémentation par défaut de `forEach`.

D'un réducteur à un itérateur

Peut-on écrire un adaptateur de type `'a iter -> 'a iterator` ?

(En espace O(1), sans construire de liste intermédiaire.)

D'un réducteur à un itérateur

Peut-on écrire un adaptateur de type `'a iter -> 'a iterator` ?

(En espace O(1), sans construire de liste intermédiaire.)

En Java ou OCaml, non.

```
let adapt (iter : 'a iter) : 'a iterator =
  let state = ref ??? in
  fun () ->
    (* We are supposed to obtain the next element. *)
    iter (fun x ->
      ???
      (* Could we please interrupt iter, return Some x,
         and resume iter the next time we are called? *)
    )
```

Scheme a call/cc, mais c'est (très) complexe.

`yield` en C# ou Python en est une version simplifiée (et moins puissante).

Pourquoi cette difficulté ?

Considérons l'itération sur un arbre binaire (cf. TD6).

Un réducteur est naturellement récursif et utilise « la pile » implicite.

- pour l'interrompre et le redémarrer, il faudrait savoir désinstaller, sauvegarder, et réinstaller une partie de la pile...

Un itérateur utilise une structure explicite pour mémoriser son état afin de pouvoir s'interrompre et reprendre plus tard au même point.

- les itérateurs du TD6 ont une liste d'arbres restant à parcourir.
- les cascades de l'amphi précédent allouent des clôtures capables de produire la suite des éléments.

D'un réducteur à un itérateur, par transformation du code

Néanmoins, si on a sous les yeux le code d'un réducteur,

- on peut manuellement le réécrire en une cascade,
- dont les clôtures chaînées forment une pile explicite.

(Suite de cette séance... cf. démo.)

Interlude

Formulation d'un parcours en profondeur d'abord comme une cascade, puis défonctionalisation pour mieux voir la pile explicite.

(Code en ligne)

(Pas dans le poly.)

À retenir

Une continuation représente la suite (suspendue) d'un calcul.

Les continuations apparaissent dans de nombreuses situations où on doit suspendre / redémarrer un calcul :

- recherche avec rebroussement ou « backtracking »
- ordonnanceur ou « scheduler »
- entrées/sorties asynchrones

Une cascade est une continuation.

Les continuations forment souvent des listes chaînées dans le tas.

Conclusion / itération

L'idée de séparer producteurs et consommateurs, via une interface standardisée qui permet de les assembler facilement, est naturelle.

Comprendre quelle interface est la meilleure est plus difficile.

- réducteurs, itérateurs, cascades, flots, ...
- ... co-routines, threads et canaux, ...

Conclusion / cours

Il semble essentiel de pouvoir :

- construire un programme en assemblant des composants
- développer chaque composant indépendamment des autres
- protéger chaque composant contre les autres
- rendre les composants interchangeables et ré-utilisables

C'est difficile, mais nous avons des outils pour cela :

- les services, c'est-à-dire les objets et fonctions ;
- l'abstraction, aux deux sens du mot ($\exists/\forall$).

Sondage

En vue de l'an prochain :

- la comparaison OCaml/Java est-elle éclairante ?
- un cours centré sur OCaml serait-il préférable ?
- comment améliorer le cours ? ...

Envoyez-moi vos suggestions et critiques constructives !

- par email à `francois.pottier@inria.fr`
- (depuis un email `@polytechnique.edu`)

Rappels

- TD aujourd'hui : compression/décompression à l'aide de flots.
- Date limite pour rendre votre projet : **ce soir** à 23h59
  - par email à `francois.pottier@inria.fr`
  - (depuis un email `@polytechnique.edu`)
- Contrôle classant le **mardi 7 juin 2016 de 9h à 12h**.