Lambda Calcul et Programmation Fonctionnelle (LCPF)

Zied Bouraoui

CRIL-CNRS & Univ Artois

bouraoui@cril.univ-artois.fr

http://www.cril.univ-artois.fr/~bouraoui/

UNIVERSITÉ D'ARTOIS

Version préliminaire du cours. Tout retour sur la forme comme sur le fond est le bienvenu.

Schémas de programmes

Introduction

Exercice : Écrivez un programme qui calcule la somme des éléments d’une liste :

```
somme l =
   case l of {
      [] -> 0;
      h:t -> h + (somme t)
}
```

Exercice : Écrivez un programme qui calcule le produit des éléments d’une liste :

```
produit l =
    case l of {
      [] -> 1;
      h:t -> h * (produit t)
}
```

Introduction

- Plusieurs programmes sur les listes se ressemblent

- Nous pouvons dire que ces programmes suivent un même **schéma**

- Une idée intéressante est donc de **les exprimer une fois pour toutes**

- Un autre intérêt d'un **schéma de programmes** est qu'il correspond à un algorithme

- Puisque plusieurs algorithmes sont possibles pour résoudre un même problème, nous pouvons changer d'algorithme en remplaçant un schéma par un autre

Schémas de programme

Les définitions de fonctions *f* précédentes, qui prennent une liste *l* en argument, suivent le même schéma récursif suivant

- Si *l* est vide alors le résultat est une valeur *a* qui ne dépend pas de *l* (cas de base).
- Sinon, soit *l* de la forme *h:t*, alors le résultat est *h* op *f(t)*, où *op* est une opération binaire (cas récursif).

```
f l=
   case l of {
      [] -> a;
      h:t -> op h (f t)
}
```

Note : Ici l'opération binaire *op* est indiquée en forme préfixée

Schémas de programme

Exercice : Soit le schéma suivant :

```
f l=
  case l of {
      [] -> a;
      h:t -> op h (f t)
}
```

Complétez le tableau :

f	a	op
Somme		
Produit		
Longueur		
Conc		

Extraction de l'opération binaire

Exercice : La fonction *somme* s'écrit comme suit en Haskell :

```
somme l =
    case l of {
     [] -> 0;
     h:t -> h + (somme t)
}
```

Donc :
f = somme, a = 0

op :

- nous avons dans la fonction : h + (somme t)
- h est l'élément courant : e
- (somme t) est le résultat accumulé : a
- nous avons h + (somme t) = e + a
- nous avons op = \ e -> \ a -> e + a

Extraction de l'opération binaire

Donc, nous pouvons réécrire la fonction somme :

```
somme l =
   case l of {
   [] -> 0;
   h:t -> (\ e -> \ a -> e + a) h (somme t)
}
```

Le cas de la fonction produit est analogue.

Extraction de l'opération binaire

La fonction longueur s'écrit comme suit en Haskell :

```
longueur l =
    case l of {
       [] -> 0;
       h:t -> 1 + (longueur t)
}
```

Donc :

f = longueur

a = 0

op :

- Nous avons dans la fonction : 1 + (longueur t)
- h est l'élément courant : e
- (longueur t) est le résultat accumulé : a
- Nous avons 1 + (longueur t) = 1 + a (e n'est pas utilisé)
- Nous avons op = \ e -> \ a -> 1 + a

Extraction de l’opération binaire

Donc, nous pouvons réécrire la fonction longueur :

```
longueur l =
   case l of {
      [] -> 0;
      h:t -> (\ e -> \ a -> 1 + a) h (longueur t)
}
```

Extraction de l'opération binaire

La fonction *conc* s'écrit comme suit en Haskell :

```
conc l1 l2 =
case l1 of {
   [] -> l2;
   h:t -> h : (conc t l2)
}
```

Donc :

f=conc

a = l2

op :

- Nous avons dans la fonction h : (conc t l2)
- h est l'élément courant : e
- (conc t l2) est le résultat accumulé : a
- Nous avons h : (conc t l2)=e : a
- Nous avons op = \ e -> \ a -> e : a

Extraction de l'opération binaire

Donc, nous pouvons réécrire la fonction conc :

```
conc l1 l2 =
     case l1 of {
     [] -> l2;
     h:t -> (\ e -> \ a -> e : a) h (conc t l2)
}
```

Schéma réduction

Le schéma réduction

- Le schéma que nous venons de voir s'appelle **réduction**.
- Nous pouvons le capturer en définissant **une fonction d'ordre supérieur**.
- La fonction **reduce** prend en **arguments** ***op***, ***a*** et **l** et calcule l'opération désirée :

```
reduce op a l =
    case l of {
    [] -> a;
    h:t -> op h (reduce op a t)
}
```

```
Prelude> reduce (\ e -> \ a -> e + a) 0 [1, 2, 3]
6
```

Nous appelons cette technique « abstraction ». Elle peut être utilisée pour définir d'autres schémas de programme.

Le schéma réduction

Haskell possède aussi la fonction (+) qui est la version préfixée et curryfiée de l'opérateur d'addition.

```
Prelude> (+) 1 2
3
```

Et même chose pour (*), (/), (:), (&&), (||), etc. Donc, nous avons :

```
Prelude> reduce (+) 0 [1, 2, 3]
6

Prelude> reduce (*) 1 [3, 4, 5]
60

Prelude> reduce (:) [4, 5, 6] [1, 2, 3]
[1, 2, 3, 4, 5, 6]

Prelude> reduce (\ e -> \ a -> 1 + a) 0 [1, 2, 3]
3
```

Le schéma réduction

Nous pouvons donc définir les fonctions vues en haut comme suit :

```
somme = reduce (+) 0
produit = reduce (*) 1
conc = reduce (:)
longueur = reduce (\ e -> \ a -> 1 + a) 0
```

Utilisation :

```
Prelude > somme [1, 2, 3]
6
Prelude > produit [3, 4, 5]
60
Prelude > longueur [1, 2, 3]
3
Prelude> conc [4, 5, 6] [1, 2, 3]
[1, 2, 3, 4, 5, 6]
```

Le schéma réduction

Quelle est le type de reduce ?

(t1 -> t2 -> t2) -> t2 -> [t1] -> t2

Schémas plier

Les schémas plier

Le résultat du schéma réduction sur une liste $l = [e_1, e_2, \ldots, e_n]$ est :

$$e_1 \text{ op } (e_2 \text{ op } \ldots(e_n \text{ op } a)\ldots)$$

Nous calculons donc d'abord e_n op a, ensuite (e_{n-1} op (e_n op a)), etc.

Mais nous pourrions aussi faire :

$$(\ldots((a \text{ op } e_1) \text{ op } e_2)\ldots) \text{ op } e_n$$

C'est-à-dire, calculer d'abord a op e_1, ensuite (a op e_1) op e_2, etc.

Le schéma plier à droite

Le premier schéma, où nous utilisons les éléments dans l'ordre inverse, s'appelle aussi **plier à droite** et correspond exactement au schéma réduction que nous avons déjà vu.

```
fold_right op a l =
    case l of {
       [] -> a;
       h:t -> op h (fold_right op a t)
}
```

Par exemple :

fold_right (+) 0 [1, 2, 3]
= 1 + fold_right (+) 0 [2, 3]
= 1 + (2 + fold_right (+) 0 [3])
= 1 + (2 + (3 + fold_right (+) 0 [])) = 1 + (2 + (3 + 0))

Le schéma plier à gauche

L'autre schéma, où on utilise les éléments dans l'ordre s'appelle **plier à gauche** et correspond à un schéma différent du schéma réduction.

Exercice : Écrivez une fonction d'ordre supérieur qui correspond au schéma **plier à gauche**.

```
fold_left op a l =
     case l of {
      [] -> a;
      h:t -> fold_left op (op a h) t
}
```

Par exemple :

fold_left (+) 0 [1, 2, 3]
= fold_left (+) (0 + 1) [2, 3]
= fold_left (+) ((0 + 1) + 2) [3]
= fold_left (+) (((0 + 1) + 2) + 3) [] = (((0 + 1) + 2) + 3)

Les schémas plier

La fonction fold_left est récursive terminale et donc plus efficace que fold_right.

Exercice : Écrivez la fonction longueur en utilisant fold_left.

Nous voulons le comportement suivant :

fold_left 0 [1, 2, 3]
= fold_left (0 + 1) [2, 3]
= fold_left ((0 + 1) + 1) [3]
= fold_left (((0 + 1) + 1) + 1) [] = (((0 + 1) + 1) + 1)

Donc, nous devons trouver la fonction op correspondante. Notez que l'accumulation est maintenant à gauche. Donc :

```
op = \ a -> \ e -> a + 1
```

```
longueur = fold_left (\ a -> \ e -> a + 1) 0
```

Schémas map

Le schéma map

Le schéma *map* consiste à **appliquer une même fonction** à tous les éléments d'une liste.

Par exemple, soit la liste $[e_1, e_2, \ldots, e_n]$ et la fonction *f*, le résultat de map est la liste

$$[f(e_1), f(e_2), \ldots, f(e_n)]$$

Exercice : Écrivez la fonction d'ordre supérieur qui correspond au schéma map :

```
map f l =
   case l of {
   [] -> [];
   h:t -> f h : (map f t)
}
```

Le schéma map

Exercice : Écrivez la fonction qui prend une liste d'entiers en argument et retourne la même liste où les entiers sont multipliés par 2.

```
fois_2 = map ((*) 2) l
```

Utilisation :

```
Prelude > fois_2 [1, 3, 4]
[2, 6, 8]
```

Le schéma map

Regardez bien le schéma map encore une fois. Cela ne vous rappelle pas quelque chose que nous avons déjà vu ?

```
map ((*) 2) [1, 2, 3]
= 2 : (map ((*) 2) [2, 3])
= 2 : (4 : (map ((*) 2) [3]))
= 2 : (4 : (6 : map ((*) 2) [])) = 2 : (4 : (6 : []))
```

Exercice : Re-écrivez la fonction qui correspond au schéma map en utilisant le schéma fold_right.

```
map f = fold_right (\ e -> \ a -> (f e) : a) []
```

Le schéma map2

Le schéma *map2* généralise le schéma *map* en utilisant deux listes comme argument.

Soit les listes $[a_1,...,a_n]$ et $[b_1,...,b_n]$ et la fonction *f*. Le résultat de map2 est la liste $[f(a_1, b_1), ..., f(a_n, b_n)]$.

Exercice : Écrivez une fonction d'ordre supérieur qui correspond à *map2* :

```
map2 f l1 l2 =
    case (l1, l2) of {
      ([], _) -> [];
      (_, []) -> [];
      (h1:t1, h2:t2) -> f h1 h2 : (map2 f t1 t2)
}
```

Le schéma *map2* est aussi appelé zip. Évidement, il est aussi possible de généraliser le schéma map à n listes.