Introduction à Caml (2) : aspects fonctionnels

1 Retour sur les définitions

1.1 Polymorphisme

Considérons la fonction

```
# let f = function x -> (x,x);;
f : 'a -> 'a * 'a = <fun>
```

La définition de **f** n'impose aucune contrainte sur le type de son argument, d'où l'introduction par Caml de la *variable de type* `'a`. Un autre exemple :

```
# snd;;
- : 'a * 'b -> 'b = <fun>
```

De telles fonctions susceptibles de s'appliquer à des objets de types différents sont dites *polymorphes*. Il en est par exemple de même de l'opérateur d'égalité.

1.2 Expressions fonctionnelles d'ordre supérieur

En Caml, les fonctions sont considérées comme des valeurs à part entière ; elles peuvent par conséquent apparaître comme arguments ou résultats d'autres fonctions. Définissons par exemple l'opérateur **h** qui à une fonction **f** associe la fonction g : $x \mapsto f(2x)$:

```
# let h = function f -> (function x -> f(2*x));;
h : (int -> 'a) -> int -> 'a = <fun>
```

Remarquer le type de **h** ... On peut ensuite appliquer cette fonction :

```
# h (function x -> float_of_int (x+3)) 5;;
- : float = 13.0
# h (function x -> string_of_int x);;
- : int -> string = <fun>
```

1.3 Nombre d'arguments d'une fonction

Considérons la fonction

```
# let somme1 x y = x+y;;
somme1 : int -> int -> int = <fun>
```

On constate que la fonction **somme1** n'a pas le même type que **somme2** définie par :

```
# let somme2(x,y) = x+y;;
somme2 : int * int -> int = <fun>
```

La fonction **somme1** a en effet *deux* arguments, tandis que la fonction **somme2** n'en a qu'*un* : un couple ![1]
Le paragraphe suivant explicite la distinction entre ces deux fonctions.

```
# somme1(2,3);;
Entree interactive:
>somme1 (2,3);;
>        ^^^
Cette expression est de type int * int,
mais est utilisee avec le type int.
```

2 Curryfication

2.1 Un peu de mathématiques

Soit $f : A \times B \to C, (x, y) \mapsto f(x, y)$ où A, B et C désignent trois ensembles. À tout élément x de A, on peut associer la fonction $f_x : B \to C, y \mapsto f_x(y) = f(x, y)$. Cela permet de définir une fonction $\phi : A \to C^B, x \mapsto f_x$ telle que $\forall (x, y) \in A \times B \quad f(x, y) = \phi(x)(y) \quad (1)$.
Réciproquement, la relation (1) permet d'associer à toute fonction ϕ de A dans C^B une unique fonction f de $A \times B$ dans C.
On dira que ϕ est la *curryfiée* [2] de f et f est la *décurryfiée* de ϕ.

2.2 Curryfication et décurryfication avec Caml

Il apparaît que les deux fonctions **somme1** et **somme2** définies précédemment sont les versions curryfiée et décurryfiée d'une même fonction somme. On peut ainsi définir une application partielle à l'aide de **somme1** :

```
# let ajoute_2 = somme1 2;;
ajoute_2 : int -> int = <fun>
# ajoute_2 8;;
- : int = 10
```

1. De la même façon, une fonction retourne une unique valeur, mais celle-ci peut être un couple ou de type `unit` ...
2. La terminologie n'a que peu de rapport avec la cuisine exotique : il s'agit ici du logicien Haskell Curry (1900-1982)

On constate ainsi que le type de **somme1**, soit `int->int->int`, résulte d'une économie de parenthèses et doit en fait se lire `int->(int->int)` (à un entier, on associe une fonction sur $\mathbb{N}$). Ce phénomène est général en Caml ; on dit que le parenthésage est associatif à droite.

2.3 Syntaxe d'une fonction curryfiée

La construction d'une fonction curryfiée à l'aide de `function` (qui n'accepte qu'un argument) est très lourde :

```
# let somme1 = function x -> function y -> x+y;;
somme1 : int -> int -> int = <fun>
```

Pour éviter d'enchaîner les `function`, Caml propose une syntaxe simplifiée à base de `fun` :

```
# let somme1 = fun x y -> x+y;;
somme1 : int -> int -> int = <fun>
```

Plus généralement, l'expression

```
let f = fun x1 x2... xn -> y
```

équivaudra à

```
let f = function x1 -> function x2 -> ... function xn -> y
```

(avec toujours l'associativité à droite du parenthésage), ou encore à

```
let f x1 x2  ... xn = y
```

Notons que la notation `fun` peut également être utilisée même lorsqu'il n'y a qu'une seule variable ...

```
# let f = let pi = 4. *. atan 1. in fun x -> x +. pi;;
f : float -> float = <fun>
```

... mais cette pratique n'est pas recommandée.
Enfin, on peut définir une fonction curryfiée en *factorisant* par un ou plusieurs arguments. Une définition de `somme1` équivalente à la précédente est :

```
# let somme1 x = function y -> x+y;;
somme1 : int -> int -> int = <fun>
```

3 Motifs et filtrage

3.1 La construction `match...with...`

Supposons que l'on veuille définir la fonction $f : \mathbb{N} \to \mathbb{N}$ telle que

$$\begin{cases} f(0) & = 0 \\ f(1) & = 5 \\ f(n) & = 2n+3 \text{ pour } n \geq 2 \end{cases}$$

Caml offre une alternative puissante à l'enchaînement de séquences `if...then...else...` :

```
# let f n =
match n with
  | 0 -> 0
  | 1 -> 5
  | n -> 2*n+3
;;
f : int -> int = <fun>
```

Le filtrage est défini par le mot-clé `match`. La fonction compare l'argument à filtrer (ici n) avec le premier motif de filtrage (ici 0). Si l'argument correspond, elle renvoie l'évaluation du membre droit, sinon elle passe au motif suivant, et ainsi de suite.

Remarques :

— L'ordre des motifs est fondamental. Par exemple,

```
# let f n =
match n with
  | 0 -> 0
  | n -> 2*n+3
  | 1 -> 5
;;
Entree interactive
>   | 1 -> 5;;
>     ^
Attention: ce cas de filtrage est inutile.
f : int -> int = <fun>
```

En effet, le calcul (faux) de $f(1)$ sera effectué à l'aide du deuxième motif, tandis que le troisième est considéré comme redondant.

— Il est possible de remplacer dans la définition de f le dernier motif « `| n -> 2*n+3` » par « `| _ -> 2*n+3` » ; le symbole `_` désigne « n'importe quoi », ce qui assure que la fonction renvoie toujours un résultat.

— Les motifs du filtrage doivent être du même type que l'argument. De même, les évaluations des membres droits doivent produire le même type.

— Il est possible de se passer de `match`, en utilisant le mot-clé `function` et en ne nommant pas la variable n :

```
# let f = function
  | 0 -> 0
  | 1 -> 5
  | n -> 2*n+3
;;
f : int -> int = <fun>
```

À noter que dans ce cas, l'identificateur n doit apparaître dans le dernier motif de filtrage car le membre droit s'y réfère.

3.2 Gardes de filtrage

Les filtrages peuvent également agir sur des couples. La fonction suivante teste si l'une des composantes est nulle (noter l'emploi de _ dans deux contextes différents) :

```
let test = function
  | (0,_) -> true
  | (_,0) -> true
  | _ -> false;;
test : int * int -> bool = <fun>
```

Supposons maintenant que nous voulions programmer la fonction caractéristique de la première bissectrice :

```
# let bissectrice = function
    | (x,x) -> 1
    |  _ -> 0;;
Entree interactive:
>    | (x,x) -> 1
>                ^
L'identificateur x est defini plusieurs fois dans ce motif.
```

La syntaxe est incorrecte. On peut y remédier en utilisant une *garde de filtrage* (à l'aide du mot clé `when`), ie une condition booléenne qui doit être vérifiée en plus du filtrage :

```
# let bissectrice = function
    | (x,y) when y=x -> 1
    |  _ -> 0;;
bissectrice : 'a * 'a -> int = <fun>
```

On peut de même définir par exemple une fonction sign : $\mathbb{R} \to \mathbb{Z}$:

```
# let sign = function
    | 0. -> 0
    | x when x>0. -> 1
    |  _ -> -1;;
sign : float -> int = <fun>
```

Remarque : filtrages non exhaustifs

La fonction précédente aurait pu être écrite sous la forme

```
# let sign = function
    | 0. -> 0
    | x when x>0. -> 1
    | x when x<0. -> -1;;
Entree interactive:
>..........function
```

```
>     | 0. -> 0
>     | x when x>0. -> 1
>     | x when x<0. -> -1.......
Attention: ce filtrage n'est pas exhaustif.
sign : float -> int = <fun>
```

Caml ne peut considérer ce filtrage comme exhaustif (il semble manquer des cas), bien qu'il le soit en réalité. Le compilateur émet donc un avertissement qui n'empêche toutefois pas d'utiliser la fonction.
Il faut en pareil cas s'efforcer de « boucher les trous »dans le filtrage. En pratique, ne jamais laisser de filtrages non exhaustifs, quitte à utiliser le symbole `_` pour le dernier cas.

Filtrage de produits cartésiens

On peut également filtrer des valeurs de type n-uplets (aussi appelés « tuples »en anglais). Par exemple :

```
# let sum_triplet = function
    | (a, b, c) -> a + b + c;;
sum_triplet : int * int * int -> int = <fun>
# let max_triplet = function
    | (a, b, c) when a >= b && a >= c -> a
    | (a, b, c) when b >= a && b >= c -> b
    | (_, _, c) -> c;;
max_triplet : 'a * 'a * 'a -> 'a = <fun>
```

On remarque qu'un motif de type `match t with (a, b, c, d) -> a+b+c+d` est *irréfutable*, c'est-à-dire qu'il fonctionne sur toutes les valeurs du type concerné. Inversement, `match t with (0, 1) -> true` est réfutable, car il échoue sur la valeur `10,10`.
On peut combiner la construction de liaison `let` avec des motifs irréfutables :

```
# let t = (1, "foo", true);;
t : int * string * bool = 1, "foo", true
# let a, b, c = t;;
a : int = 1
b : string = "foo"
c : bool = true
```

ce qui permet de manipuler aisément les produits cartésiens. Nous verrons plus tard que le filtrage s'étend à d'autres types de Caml, et notamment aux listes et arbres.

4 Fonctions récursives

4.1 La contruction `let rec...`

Une fonction peut être amenée à s'appeler elle-même pour évaluer un calcul. Elle est alors dite *récursive* et doit être définie en Caml à l'aide des mots clés `let rec`. Par exemple, la fonction factorielle peut être définie sur $\mathbb{N}$ par :

```
# let rec factorielle = function
    | 0 -> 1
    | n -> n*factorielle(n-1);;
factorielle : int -> int = <fun>
```

À noter que le filtrage commence par étudier les *cas terminaux* pour lesquels la fonction renvoie explicitement un résultat.
Il est possible de suivre les appels récursifs de la fonction :

```
# trace "factorielle";;
La fonction factorielle est dorenavant tracee.
- : unit = ()
# factorielle 4;;
factorielle <-- 4
factorielle <-- 3
factorielle <-- 2
factorielle <-- 1
factorielle <-- 0
factorielle --> 1
factorielle --> 1
factorielle --> 2
factorielle --> 6
factorielle --> 24
- : int = 24
```

4.2 Fonctions mutuellement récursives

On utilise le mot-clé `and` (`rec` n'ayant pas à être répété). Par exemple, une manière totalement inefficace de tester la parité d'un entier consiste à écrire :

```
# let rec est_pair = fun
    | 0 -> true
    | n -> est_impair (n-1)
  and
  est_impair = fun
    | 0 -> false
    | n -> est_pair (n-1);;
est_pair : int -> bool = <fun>
est_impair : int -> bool = <fun>
```

4.3 Préconditions et Post-conditions

Afin de s'assurer de la correction des fonctions (c'est-à-dire qu'elles calculent bien ce qu'on attend d'elles), il est nécessaire de les prouver mathématiquement. Nous n'aborderons pas les notions formelles (logique de Hoare, notamment), mais il reste nécessaire de fournir une justification. Pour cela, il est très utile de spécifier, pour chaque fonction non triviale, deux choses :

sa précondition : une propriété que doivent vérifier ses arguments (par exemple le fait d'être positifs, ou que deux tableaux doivent avoir la même longueur)

sa post-condition : une propriété que doit vérifier la valeur de retour de la fonction. Cette propriété dépend généralement des arguments !

Reprenons l'exemple de la factorielle :

```
# let rec factorielle n =
    (* precondition: n >= 0 *)
    match n with
    | 0 -> 1
    | _ -> n * factorielle (n-1)
    (* postcondition: factorielle n = n! *)
;;
factorielle : int -> int = <fun>
```

Pour véritablement prouver que la post-condition est vérifiée par le code de `factorielle`, il faut raisonner par *récurrence* sur l'argument n (le cas terminal correspondant au cas de base de la récurrence). Nous verrons d'autres formes de raisonnements dits *inductifs* qui permettent de prouver les fonctions récursives.