Expression Logique et Fonctionnelle . . . Évidemment

TP n° 4 : Types sommes. Filtrage. λ-calcul.

Objectifs du TP
- Déclarations de types.
- Types sommes, enregistrements.
- Définition de fonctions par filtrage.
- λ-calcul en CAML : la syntaxe.

1 Déclaration de types

Comme beaucoup d'autres langages de programmation, OCAML offre au programmeur la possibilité de définir ses propres types de données à partir des types de base. Pour cela, il faut utiliser le mot clé `type` suivi du nom du type à définir et de la définition de ce type.

```
# type identificateur = string ;;
type identificateur = string
```

Il est même possible de déclarer simultanément plusieurs types, leurs définitions faisant référence à d'autres types en cours de déclaration.

```
# type identificateur = string
  and valeur = int
  and variable = identificateur * valeur
  and environnement = variable list ;;
type identificateur = string
and valeur = int
and variable = identificateur * valeur
and environnement = variable list
```

Supposons que nous voulions définir une fonction qui à partir d'un identificateur, d'une valeur et d'un environnement construit un nouvel environnement dans lequel une variable a été ajoutée, c'est–à–dire une fonction dont le type est

`identificateur` $->$ `valeur` $->$ `environnement` $->$ `environnement`.

Nous pourrions écrire :

```
let ajoute x v e = (x,v)::e ;;
```

Malheureusement, cette définition donne une fonction dont le type est plus général que celui désiré :

```
# let ajoute x v e = (x,v)::e ;;
val ajoute : 'a -> 'b -> ('a * 'b) list -> ('a * 'b) list = <fun>
```

Cela provient du mécanisme de type qui ne peut pas déduire de la définition de la fonction que `x` doit être de type `identificateur`, etc... même si c'est pourtant la volonté du programmeur.

Une solution consiste à laisser le soin au programmeur de typer la fonction qu'il définit :

```
# let ajoute (x : identificateur)
             (v : valeur)
             (e : environnement) : environnement =
                (x,v)::e ;;
val ajoute : identificateur -> valeur -> environnement -> environnement = <fun>
```

ou plus simplement encore

```
# let ajoute x v e : environnement = (x,v)::e ;;
val ajoute : identificateur -> valeur -> environnement -> environnement = <fun>
```

Il existe d'autres solutions pour permettre une reconnaissance syntaxique du type des expressions lorsque ce type est définit par le programmeur, sans avoir recours au typage explicite : les types sommes et les enregistrements.

2 Types sommes

Un *type somme* est un type déclaré par énumération des différentes *formes* que peuvent prendre les valeurs de ce type. Les formes sont déclarées au moyen de *constructeurs* dont les identificateurs en CAML doivent nécessairement débuter par une lettre majuscule.

Ainsi, le programmeur peut (s'il en éprouve le besoin) déclarer un type `booleen` n'ayant que deux valeurs `Vrai` et `Faux`.

```
# type booleen = Vrai | Faux ;;
type booleen = Vrai | Faux
# Vrai ;;
- : booleen = Vrai
# Faux ;;
- : booleen = Faux
```

Une telle déclaration de type correspond à ce qu'on appelle dans d'autres langages de programmation à un type énuméré. Le nombre de valeurs de types ainsi déclarés est fini.

En CAML, les types sommes ne se limitent pas à des types finis. Les constructeurs ne sont pas nécéssairement constants, mais peuvent être paramétrés par d'autres types. Par exemple, le type `int_etendu` étend le type `int` avec un constructeur non paramétré `Non_defini`, et un constructeur, `Int`, paramétré par les `int`.

```
# type int_etendu = Non_defini | Int of int ;;
type int_etendu = Non_defini | Int of int
# Non_defini ;;
- : int_etendu = Non_defini
# Int 3 ;;
- : int_etendu = Int 3
# Int (3+2) ;;
- : int_etendu = Int 5
```

Question 1 Programmez la fonction `quotient` de type `int->int->int_etendu` qui donne le quotient entier des deux entiers passés en paramètres lorsque le second n'est pas nul, et la valeur `Non_defini` dans le cas contraire. On doit avoir

```
# quotient 17 3 ;;
- : int_etendu = Int 5
# quotient 17 0 ;;
- : int_etendu = Non_defini
```

Les types sommes permettent même de définir des structures de données récursives. Par exemple, pour définir un type pour les listes d'entiers, si on se souvient qu'une liste d'entiers est soit la liste vide, soit un couple dont le premier élément est un entier et le second une liste d'entiers, on peut écrire la déclaration :

```
# type liste_entiers = Vide | Cons of (int*liste_entiers) ;;
type liste_entiers = Vide | Cons of (int * liste_entiers)
# Vide ;;
- : liste_entiers = Vide
# Cons (1,Vide) ;;
- : liste_entiers = Cons (1, Vide)
```

Question 2 Construisez en CAML une valeur de type `liste_entiers` représentant la liste $(1, 2, 3, 1)$.

3 Filtrage

Comment peut-on définir des fonctions dont les paramètres sont d'un type somme ? Par exemple, comment définir les opérateurs logiques usuels sur notre type `booleen` ? Comment définir l'arithmétique usuelle sur notre type `int_etendu` ? Comment calculer la longueur d'une liste ?

Lorsque le type somme ne contient que des constructeurs d'arité 0 (ou encore non paramétrés), une simple énumération des cas peut suffire. Par exemple, on peut programmer l'opérateur `et` en écrivant

```
# let et a b =
    if a = Vrai then
```

```
      b
    else
      Faux ;;
val et : booleen -> booleen -> booleen = <fun>
# et Vrai Faux ;;
- : booleen = Faux
# et Vrai Vrai ;;
- : booleen = Vrai
# et Faux Vrai ;;
- : booleen = Faux
# et Faux Faux ;;
- : booleen = Faux
```

Mais ce style de programmation ne convient plus dès qu'un des constructeurs du type a une arité au moins égale à 1. En effet comment extraire la valeur du paramètre d'un tel constructeur ?

La réponse réside dans le *filtrage de motifs* (*pattern matching* en anglais). En CAML, c'est l'expression `match ... with ...` qui permet d'effectuer du filtrage de motifs. Sa syntaxe est

```
match expression with
  | motif_1 -> expr_1
  | motif_2 -> expr_2
  ...
  | motif_n -> expr_n
```

Voici comme premier exemple, une définition de notre opérateur logique `et` à l'aide du filtrage des quatre motifs possibles pour les deux booléens.

```
let et a b =
   match a,b with
    | Vrai, Vrai -> Vrai
    | Vrai, Faux -> Faux
    | Faux, Vrai -> Faux
    | Faux, Faux -> Faux ;;
```

En utilisant le motif universel _, satisfait par toutes les instances, on peut écrire de manière plus concise :

```
let et a b =
   match a,b with
    | Vrai, Vrai -> Vrai
    | _ -> Faux ;;
```

ce qui signifie : lorsque les deux paramètres sont `Vrai` alors on renvoie `Vrai` et dans tous les autres cas on renvoie `Faux`.

Remarque 1 : La première version de la fonction `et` énumère quatre motifs deux à deux incompatibles, tandis que la deuxième version ne donne que deux motifs non incompatibles : le second (motif universel) contient le premier. L'incompatibilité ou non des motifs dans une énumération d'un filtrage a une conséquence sur l'ordre d'écriture des motifs. Dans le premier cas (incompatibilité des motifs), l'ordre importe peu. Dans le second cas (non incompatibilité des motifs), l'ordre est important. Par exemple, si on écrit

```
# let et a b =
     match a,b with
      | _ -> Faux
      | Vrai, Vrai -> Vrai ;;
Warning U: this match case is unused.
# et Vrai Vrai =
- : booleen = Faux
```

nous pouvons remarquer un message d'avertissement de l'interprète qui prévient qu'un motif est inutile. De plus la fonction ainsi définie est fausse.

Remarque 2 : Si l'énumération des différents cas d'un filtrage n'est pas exhaustif, l'interprète avertit le programmeur en donnant un exemple de motif non couvert par l'énumération.

```
# let et a b =
    match a,b with
     | Vrai, Vrai -> Vrai
     | Vrai, Faux -> Faux
     | Faux, Vrai -> Faux ;;
Warning P: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
(Faux, Faux)
val et : booleen -> booleen -> booleen = <fun>
```

Remarque 3 : L'expression `match` peut être utilisée dans tous contexte. Dans les exemples qui précèdent, elle est utilisée dans le contexte d'une définition de fonctions à deux paramètres, le filtrage portant sur les deux paramètres. Dans le contexte d'une définition d'une fonction à un seul paramètre, une autre écriture du filtrage de motif est possible qui n'utilise pas l'expression `match` :

```
let une_fonction = function
 | motif_1 -> expr_1
 | motif_2 -> expr_2
 ...
 | motif_n -> expr_n
```

En fait toutes les fonctions que l'on définit en OCaml le sont par filtrage de motifs.

Question 3 Programmez la négation avec du filtrage de motifs sans utiliser l'expression `match`.

Voyons maintenant comment par le filtrage de motifs, on peut extraire des parties d'une valeur d'un type somme. Nous allons l'illustrer sur la fonction calculant la somme de deux `int_etendu`. Si les deux valeurs sont de la forme `Int n` et `Int p` alors la somme est de la forme `Int (n+p)`. Dans le cas contraire, la somme est `Non_definie`.

```
let plus n1 n2 =
   match n1,n2 with
    | Int n, Int p -> Int (n+p)
    | _ -> Non_defini
```

Question 4 Reprenez la fonction `quotient` pour l'étendre aux `int_etendu`. Le type de la fonction doit maintenant être

`int_etendu` $->$ `int_etendu` $->$ `int_etendu`.

Question 5 Définissez une fonction calculant la longueur d'une liste d'entiers. Son type est

`liste_entiers` $->$ `int`.

```
# longueur Vide ;;
- : int = 0
# longueur (Cons (1,Vide)) ;;
- : int = 1
# longueur (Cons (2, Cons (1,Vide))) ;;
- : int = 2
```

Remarque 4 : Dans un motif, on ne peut pas utiliser plusieurs occurences de la même variable de motif. Par exemple, pour définir l'égalité de deux `int_etendu` on pourrait être tenté d'écrire

```
let egal n1 n2 =
  match n1,n2 with
   | Int n, Int n -> Vrai
   | Non_defini, Non_defini -> Vrai
   | _ -> Faux ;;
```

mais ce code produit le message d'erreur `Variable n is bound several times in this matching`.

Si on tient malgré tout à suivre ce schéma de motif, on peut écrire

```
let egal n1 n2 =
  match n1,n2 with
    | Int n, Int p -> if n=p then Vrai else Faux
    | Non_defini, Non_defini -> Vrai
    | _ -> Faux ;;
```

ou bien utiliser la notion de *garde*

```
let egal n1 n2 =
  match n1,n2 with
    | Int n, Int p when n=p  -> Vrai
    | Non_defini, Non_defini -> Vrai
    | _ -> Faux ;;
```

Question 6 Définissez une fonction de type `int->liste_entiers->int_etendu` qui donne le rang de l'entier passé en premier paramètre dans la liste passée en second paramètre si cet entier y figure, qui donne la valeur `Non_defini` dans le cas contraire.

```
# rang 1 Vide ;;
- : int_etendu = Non_defini
# rang 1 (Cons (1,Vide)) ;;
- : int_etendu = Int 0
# rang 1 (Cons (2, Cons (1,Vide))) ;;
- : int_etendu = Int 1
```

4 λ-calcul

Le but de cette partie est de débuter l'implantation du λ-calcul en CAML. On ne s'occupe ici que de la partie syntaxique, la réduction et le calcul de forme normale étant reporté à un TP ultérieur.

On rappelle la grammaire définissant les termes (ou programmes) de Core ML.

$\langle\lambda\text{-terme}\rangle$	$::=$	$\langle\text{variable}\rangle$	(variable)
	$\mid$	$(\lambda\langle\text{variable}\rangle.\langle\lambda\text{-terme}\rangle)$	(abstraction)
	$\mid$	$(\langle\lambda\text{-terme}\rangle\ \langle\lambda\text{-terme}\rangle)$	(application)
$\langle\text{variable}\rangle$	$::=$	$x \mid y \mid z \ \ldots$	

Pour représenter les λ-termes on peut déclarer le type suivant :

```
type lambda_terme =
 | Var of string                                (* variables *)
 | Abstr of (string * lambda_terme)             (* abstraction *)
 | Appl of (lambda_terme * lambda_terme)        (* application *)
```

Question 7 Définissez les λ-termes suivant en CAML selon ce type :

1. $t_1 \equiv x$;
2. $t_2 \equiv (x\ y)$;
3. $t_3 \equiv \mathbf{I} \equiv \lambda x.x$;
4. $t_4 \equiv \mathbf{K} \equiv \lambda xy.x$;
5. $t_5 \equiv \mathbf{K}_* \equiv \lambda xy.y$;
6. $t_6 \equiv \mathbf{S} \equiv \lambda xyz.(x\ z\ (y\ z))$;
7. $t_7 \equiv \mathbf{\Omega} \equiv (\lambda x.(x\ x))\ (\lambda x.(x\ x))$;
8. $t_8 \equiv \mathbf{Y} \equiv \lambda f.(\lambda x.(f\ (x\ x))\ \lambda x.(f\ (x\ x)))$.

Question 8 Réalisez une fonction de conversion d'un λ-terme en une chaîne de caractères

`lambdaterme_en_chaine: lambda_terme -> string`.

Cette fonction permet de passer de la représentation interne des λ-termes (type `lambda_terme` en CAML) à une représentation externe plus lisible. Les variables seront transformées par leur propre nom. Les applications seront parenthésées. Les abstractions seront notées avec le symbole \ suivi immédiatement du nom de la variable d'abstraction.

Voici quelques exemples obtenus avec cette fonction :

```
# lambdaterme_en_chaine t2 ;;
- : string = "(x y)"
# lambdaterme_en_chaine t3 ;;
- : string = "\\x.x"
# lambdaterme_en_chaine t4 ;;
- : string = "\\x.\\y.x"
# lambdaterme_en_chaine t6 ;;
- : string = "\\x.\\y.\\z.((x z) (y z))"
# lambdaterme_en_chaine t8 ;;
- : string = "\\f.(\\x.(f (x x)) \\x.(f (x x)))"
```

Attention, le caractère \ est un caractère spécial (caractère d'échappement). Il faut le doubler pour le considérer comme un caractère ordinaire.

En utilisant la fonction précédente et la fonction `Format.pp_print_string` on peut réaliser un imprimeur (ou pretty-printer) pour les λ-termes. En voici le code CAML.

```
let imprimer_lambdaterme f t =
  Format.pp_print_string f (lambdaterme_en_chaine t)
```

On peut installer cet imprimeur dans la table des imprimeurs de l'interpréteur avec la directive `#install_printer`.

```
# #install_printer imprimer_lambdaterme ;;
```

Une fois cette directive donnée, tous les λ-termes seront imprimés avec notre imprimeur.

```
# t1 ;;
- : lambda_terme = x
# t2 ;;
- : lambda_terme = (x y)
# t3 ;;
- : lambda_terme = \x.x
# t4 ;;
- : lambda_terme = \x.\y.x
# t6 ;;
- : lambda_terme = \x.\y.\z.((x z) (y z))
# t7 ;;
- : lambda_terme = (\x.(x x) \x.(x x))
```

Question 9 Réalisez deux fonctions qui calculent l'ensemble des variables libres et l'ensemble des variables liées d'un λ-terme, ces ensembles étant de type `string list`.