PFITA – Cours/TD 1–2

UPS – L3 Info – Semestre 5

Types simples et expressions

() (1) est l'unique valeur du type `unit`. `false` et `true` sont les 2 valeurs du type bool (2).

'a s'appelle une variable de type (3) et désigne n'importe quel type (4).

Pour que l'expression « `if` e_0 `then` e_1 `else` e_2 » soit bien typée, il faut que e_0 soit un bool (e_0 : bool) (5) et que e_1 et e_2 soient de types compatibles (6).

5 a pour type int (7), 5.0 a pour type float (8), 'a' a pour type char (9).

Définitions locales et globales

Une définition locale s'écrit sous la forme « `let x =` $expr_1$ `in` $expr_2$ » où `x` est un identificateur auquel va être lié (temporairement) la valeur de $expr_1$ avant d'évaluer $expr_2$.

Une définition locale simultanée s'écrit sous la forme « `let x =` e_1 `and y =` e_2 `in` e_3 ».
Quelle(s) différence(s) y a-t-il avec « `let x =` e_1 `in let y =` e_2 `in` e_3 » ?

La différence se situe dans la variable temporaire x.
Sa portée n'est pas la même dans les deux phases.
Il y a une simultanéité dans le cas 1. (10)

Une définition globale s'écrit sous la forme « `let x = expr` ». L'expression `expr` est évaluée puis liée à l'identificateur `x` dans l'environnement global.
Une définition globale simultanée s'écrit « `let` x_1 `=` $expr_1$ `and` x_2 `=` $expr_2$ ».

Fonctions à un ou plusieurs arguments

Une expression fonctionnelle est de la forme « `fun` *x* `->` *corps* ».
Son type est de la forme « *type_de_x* `->` *type_du_corps* ».

Quel est le type de « `fun x -> x +. 1.` » ?

_ : float -> float (11)

Fonction à 2 arguments = fonction qui prend 1 arg. et renvoie une fonction (qui prend le 2e arg...)
Quel est le type de « `fun x -> fun y -> x <> y` » ?

_ : 'a -> 'a -> bool (12)

Les écritures suivantes sont équivalentes :

- `let f = fun x -> (fun y -> e)`
- `let f = fun x -> fun y -> e`
- `let f = fun x y -> e`
- `let f x y = e`

Pour appliquer une fonction `f` à 2 arguments a_1 et a_2, on écrit « `f` a_1 a_2 » qui équivaut à « (`f` a_1) a_2 ».
On dit que l'application de fonctions est associative par la gauche.

Fermeture et curryfication

Une fermeture est valeur représentant une fonction (13). Elle est composée de son code et de son environnement de définition (14) (= ensemble des valeurs des variables libres qui ont été capturées au moment de la définition).

En appliquant une fermeture à un arg., OCaml construit l'environnement d'appel (15) (= l'environnement de définition (16) + liaison des paramètres effectifs aux paramètres formels).

Une fonction à 2 arguments ou plus s'appelle une fonction curryfiée. On peut ne lui passer qu'un argument. En faisant cela on parle d'application partielle (17).

Donner 1 exemple de fermeture capturant une variable définie globalement.

```
let glob = 1;;
let f = fun x -> glob + x;;
```

(18)

Donner 1 exemple de fermeture obtenue par application partielle.

```
let axpb = fun a b x -> a * x + b;;
let g = axpb 2 1;;
val axpb : int -> int -> int -> int = <fun>
val g : int -> int = <fun>
```

(19)

Types produit

Pour construire un n-uplet, on écrit $(a_1, a_2, \ldots, a_n)$ où a_1, a_2, etc. ne sont pas nécessairement du même type.

`(true, 1 + 1, 3)` est un triplet (20) du type `bool * int * int` (21)

`(true, (1 + 1, 3))` est un couple (22) du type `bool * (int * int)` (23)

Les n-uplets sont pratiques pour construire des fonctions qui renvoient plusieurs valeurs.

Ils peuvent aussi être utilisés en argument.

`fun x -> fun y -> x + y` est une fonction curryfiée. Application partielle possible (24).
type `int -> int -> int`

`fun (x, y) -> x + y` est une fonction décurryfiée. Application partielle impossible (25).
type `int * int -> int`

Filtrage par motifs

Filtrage par motifs = *pattern matching*.

En représentant les nombres complexes par un couple de flottants, définir une fonction **addc** effectuant l'addition de deux nombres complexes.

= ligne du match

v1
```
let addc = fun (a,b) (c,d) -> (a+.c, b+.d);;
```
filtrage extrêmement avec 1 seule branche

v2
```
let addc z1 z2 = match z1, z2 with (a,b), (c,d) -> (a+.c, b+.d)
```
pas d'erreur

v3
```
let addc z1 z2 = let (a,b) = z1 in let (c,d) = z2 in (a+.c, b+.d)
```

(26)