Algorithmique - Programmation 1

Cours 4

Université Henri Poincaré

CESS Epinal

Automne 2008

Plan

Rappels : Types en Caml

Polymorphisme

Le filtrage

Le type produit cartésien

Théorème de Curry

Application partielle

Rappels : Types en Caml

- Types de base :
  - booléens (`bool`)
  - entiers relatifs (`int`)
  - réels (`float`)
  - caractères (`char`)
  - chaînes (`string`)

- Types fonctionnels :
  - `T1 -> T2`
    (exemple: `val float_of_int: int -> float = <fun>`)

  NB: `T1` ou `T2` peuvent eux-même être des types fonctionnels.

Rappels : Types en Caml (suite)

- Quelques exemples de types fonctionnels :

```
# let f = function x -> x;;
```

Rappels : Types en Caml (suite)

- Quelques exemples de types fonctionnels :

```
# let f = function x -> x;;
val f :  'a -> 'a = <fun>
```

Rappels : Types en Caml (suite)

- Quelques exemples de types fonctionnels :

```
# let f = function x -> x;;
val f :  'a -> 'a = <fun>

# let g = function a -> function b -> function c ->
        (b (a c));;
```

Rappels : Types en Caml (suite)

- Quelques exemples de types fonctionnels :

```
# let f = function x -> x;;
val f :  'a -> 'a = <fun>

# let g = function a -> function b -> function c ->
        (b (a c));;
val g :  ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c =
<fun>
```

Rappels : Types en Caml (suite)

- Quelques exemples de types fonctionnels :

```
# let f = function x -> x;;
val f :  'a -> 'a = <fun>

# let g = function a -> function b -> function c ->
        (b (a c));;
val g :  ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c =
<fun>

# let a b c = (b c);;
```

Rappels : Types en Caml (suite)

- Quelques exemples de types fonctionnels :

```
# let f = function x -> x;;
val f :  'a -> 'a = <fun>

# let g = function a -> function b -> function c ->
        (b (a c));;
val g :  ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c =
<fun>

# let a b c = (b c);;
val a :  ('a -> 'b) -> 'a -> 'b = <fun>
```

Rappels : Types en Caml (suite)

- Quelques exemples de types fonctionnels :

```
# let f = function x -> x;;
val f :  'a -> 'a = <fun>

# let g = function a -> function b -> function c ->
        (b (a c));;
val g :  ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c =
<fun>

# let a b c = (b c);;
val a :  ('a -> 'b) -> 'a -> 'b = <fun>
```

- Fonctions polymorphes : le type des données manipulées est déterminé dynamiquement (*cf* suite du cours)

Plan

Rappels : Types en Caml

Polymorphisme

Le filtrage

Le type produit cartésien

Théorème de Curry

Application partielle

Polymorphisme

- Ambiguïté sur le type que peut prendre une valeur

  Exemple :

```
# let s = function f -> function g -> function x ->
          (f x) + (g x);;
```

Polymorphisme

- Ambiguïté sur le type que peut prendre une valeur

  Exemple :

```
# let s = function f -> function g -> function x ->
          (f x) + (g x);;
val s :  'a -> int -> 'a -> int -> 'a -> int =
<fun>
```

- `'a` est une variable de type qui sera *instanciée* à l'appel de la fonction `f`
- Si plusieurs valeurs sont ambigues, `Caml` utilisera les variables de type `'a`, `'b`, `'c`, *etc*

Polymorphisme (suite)

Définition (Polymorphisme)

En programmation, le polymorphisme réfère à la propriété d'un langage de proposer la définition de types de données génériques reposant sur une interface uniforme.

- Le polymorphisme permet d'écrire un code plus concis
- et aussi réutilisable puisque plus général
- Illustration : développement de bibliothèques (*libraries*)
- Exemple de fonction polymorphe en `Caml` :

```
# max;;
- : 'a -> 'a -> 'a = <fun>
```

Plan

Rappels : Types en Caml

Polymorphisme

Le filtrage

Le type produit cartésien

Théorème de Curry

Application partielle

Le filtrage

- Appelé aussi *pattern matching*
- Notion de définition par cas introduite en cours précédemment
- En `Caml`, s'écrit au moyen des mots-clés `match ... with`
- Exemple :

```
let rec f = function x -> match x with
                        0 -> 1
                      | 1 -> 1
                      | n -> (f (n-1)) + (f (n-2));;
val f : int -> int = <fun>
```

Le filtrage (suite)

- Les définitions ci-dessous sont équivalentes

```
let g = function 0 -> 1
                 | n -> n * (f (n-1));;

let g = function x -> match x with
                        0 -> 1
                      | n -> n * (f (n-1));;

let g x = match x with 0 -> 1
                     | n -> n * (f (n-1));;
```

Plan

Rappels : Types en Caml

Polymorphisme

Le filtrage

Le type produit cartésien

Théorème de Curry

Application partielle

Le type produit cartésien

Définition (Produit cartésien)

Structure de données permettant de représenter les n-uplets.

- Exemple : $x = (v_1, v_2, \ldots, v_n)$
- NB: si $\forall i \in [1..n]$, $v_i \in E$ alors

$$x \in \overbrace{E \times \cdots \times E}^{n \text{ fois}}$$

- En `Caml`, un élément d'un produit cartésien est construit au moyen de la virgule
  Exemple :

```
# let a = (3, 2.5);;
val a :  int * float = (3, 2.5)
```

Le type produit cartésien (suite)

- Les éléments d'un n-uplet peuvent être de types différents
- En `Caml`, les n-uplets ont une dimension *finie*
- Dans certains langages, on appel les valeurs de type produit cartésien des `record` ou encore `tuples`
- Les produits cartésiens permettent de représenter des données complexes (coordonnées du plan, de l'espace, horaires, nombres complexes, vecteurs, *etc*)

Le type prorduit cartésien (suite)

- Une fonction peut prendre des paramètres de type produit cartésien
  Exemple :

```
# let c = function (h,m,s) -> h * 3600 + m * 60 + s;;
val c : int * int * int -> int = <fun>

# let d (h,m,s) = h * 3600 + m * 60 + s;;
val d : int * int * int -> int = <fun>
```

- Exercice : écrire une fonction permettant de faire la somme de vecteurs de $\mathbb{R}^3$

Le type prorduit cartésien (suite)

- Le pattern matching s'applique aussi sur des données de type produit cartésien

  Exemple :

```
# let f = function x -> match x with
              (0., 0.) -> 0.
            | (a , b ) -> sqrt( a*.a +. b*.b );;
```

- Typage d'une expression utilisant le type produit cartésien

```
# let somme = function x -> function y -> x + y;;

# ((somme 2 4), (somme 1), somme);;
```

  $\longrightarrow$ inférence de type sur chaque dimension

Le type prorduit cartésien (suite)

- Cas particulier du produit cartésien : les *couples* (appelés aussi *paires*)
- Fonctions spécifiques prédéfinies en `Caml` :

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

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

- NB : en théorie, tout n-uplet peut s'écrire sous forme de couples imbriqués

Plan

Rappels : Types en `Caml`

Polymorphisme

Le filtrage

Le type produit cartésien

Théorème de Curry

Application partielle

Théorème de Curry

- Pour les fonctions à plusieurs paramètres, nous avons vu 2 notations :

```
# let f1 = function x -> function y -> x + y;;
val f1 : int -> int -> int = <fun>

# let f2 = function (x,y) -> x + y;;
val f2 : int * int -> int = <fun>
```

- Quelle notation privilégier ?

$$\begin{array}{rcccc} f_1 & : & E \times F & \longrightarrow & G \\ & & (x, y) & \longmapsto & f(x, y) \end{array}$$

$$\begin{array}{rcccccc} f_2 & : & E & \longrightarrow & F & \longrightarrow & G \\ & & x & \longmapsto & y & \longmapsto & (f\ \ x\ \ y) \end{array}$$

Théorème de Curry (suite)

- D'après le théorème de Curry, ces 2 formes sont équivalentes

Définition (Currification)

Technique transformat une fonction prenant un n-uplet en entrée en une composition de fonctions prenant chacune un seul argument en entrée.

- Ce théorème peut se voir comme un isomorphisme entre fonctions (isomorphisme de Curry-Howard)
- NB: tout langage utilisant la notion de fermeture peut manipuler des fonctions currifiées

Théorème de Curry (suite)

- Currification / décurrification en Caml :

```
# let curry   = function f ->
                function x -> function y -> f (x,y);;

# let uncurry = function f ->
                function pair ->
                          (f (fst pair) (snd pair));;
```

- Typage de ces fonctions ?
- Intérêt de la currification :
  - application fonctionnelle ramenée à un mécanisme uniforme (évaluation d'un paramètre)

Plan

Rappels : Types en Caml

Polymorphisme

Le filtrage

Le type produit cartésien

Théorème de Curry

Application partielle

Application partielle

Définition (Application partielle)

Appel d'une fonction à n paramètres en ne fixant la valeur que des m premiers paramètres ($m \leq n$).

- Exemple :

```
# let f = function x -> function y -> x + y;;

# let g = function u -> (f u);;
```

- Une application totale retourne une valeur constante (valeur de la fonction pour une valeur donnée de l'ensemble de ses paramètre $\approx$ valeur en un point)
- Une application partielle retourne une valeur fonctionnelle

Application partielle (suite)

- Exemple :

```
# let d = function f -> function x ->
          let dx = x /. (10e+4) in
          ((f (x +. dx)) -. (f x)) /. dx;;

# let dd = function f -> (d (d f));;
```

- Typez ces fonctions et testez les au moyen des valeurs suivantes :

```
# let f = function x -> x *. x *. x;;

# let x = 2. ;;
```

- Conclusion ?