Cours 2 – Types algébriques

Programmation fonctionnelle
CFA INSTA - Master 1 - Analyste Développeur

Pierre Talbot (pierre.talbot@univ-nantes.fr)

Université de Nantes

10 avril 2019

UNIVERSITÉ DE NANTES

Le menu

Types algébriques

$\langle T, U, \ldots \rangle$ ::=
- | ...
- | `(`T`,` U`)`

Types Ty_3
(types Ty_2)
(tuples)

$\langle D \rangle$ ::=
- | `type t =` T
- | `type t = { x :` T `; y :` U `}`
- | `type t =` **`VarT`** `of` T `|` **`VarU`** `of` U

Déclaration de types D_3
(alias de type)
(enregistrement)
(type somme)

Expression de types (K_3)

$\langle n, m, p, q \ldots \rangle ::=$	**Expression**
...	(K_1, K_2)
$(n : T)$	(annotation de types)
(n, m)	(tuple)
{ **x** = n ; **y** = m }	(enregistrement)
v.x	(accès champs d'un enregistrement)
Var n	(variante)
`match` n `with` m `->` p `|` m' `->` q	(pattern matching)

Note : les tuples et enregistrements sont n-aires (peuvent contenir plus de deux éléments).

Le menu

Tuple

Un tuple est une collection de valeurs de types différents dont la taille est connue à la compilation.

```
type coordinate = int * int

let make_coord x y = (x, y)

let translate_x coord d =
  let (x, y) = coord in
  (x + d, y)
```

On peut *déstructurer* (via *pattern matching*) un tuple avec un `let` pour récupérer ses éléments.

Tuple : pattern matching

On peut directement déstructurer un tuple dans le paramètre de la fonction :

```
let translate_x coord d =
  let (x, y) = coord in
  (x + d, y)
```

$\rightarrow$

```
let translate_x (x, y) d =
  (x + d, y)
```

Tuple : fst et snd

On peut ignorer un composant d'un tuple lors du *pattern matching* :

```
let (x, _) = make_coord 4 2
```

Les fonctions `fst` et `snd` sont déjà définies de la sorte :

```
let fst (x, _) = x
let snd (_, y) = y
```

- L'élément _ permet d'ignorer certaines valeurs qui ne nous intéressent pas.

Le menu

Enregistrement

Un enregistrement est un tuple dont les éléments sont labellisés.

```
type coord = {
  x: int;
  y: int;
}

let make_coord x y = {x=x; y=y}

let translate_x coord d =
  {x=(coord.x + d); y=coord.y}
```

Enregistrement : pattern matching

Comme un tuple, on peut directement déstructurer un enregistrement dans le paramètre de la fonction :

```
let translate_x {x=x; y=y} d =
  {x=(x + d); y}
```

Si on veut modifier juste un élément de l'enregistrement et répéter le reste :

```
let translate_x coord d =
  {coord with x=(coord.x + d)}
```

Le menu

Type somme

- Un type somme permet de construire des disjonctions de types.
- Intuition : quand on veut dire “je retourne soit A, soit B.”.
- Par exemple “je retourne -1 si je ne trouve pas la valeur dans cette liste, et la valeur sinon”.
- Les types somme permettent de faire ça proprement !

```
type color =
    Red
  | Green
  | Blue
```

Type somme : pattern matching

On peut facilement tester la valeur d'un élément de type somme :

```
type color = Red | Green | Blue

let color_to_string color =
  if color = Red then "red"
  else if color = Green then "green"
  else "blue"
```

Mais imaginons qu'on rajoute une nouvelle couleur... Problème ?

Type somme : pattern matching

On peut facilement tester la valeur d'un élément de type somme :

```
type color = Red | Green | Blue

let color_to_string color =
  if color = Red then "red"
  else if color = Green then "green"
  else "blue"
```

Mais imaginons qu'on rajoute une nouvelle couleur... Problème ?
Oui, car la fonction `color_to_string` sera toujours valide et renverra `blue` pour cette nouvelle couleur.

Type somme : pattern matching

On utilise généralement la construction `match` pour déstructurer un type somme :

```
type color = Red | Green | Blue

let color_to_string color =
  match color with
  | Red -> "red"
  | Green -> "green"
  | Blue -> "blue"
```

Si on rajoute une nouvelle couleur, le compilateur ne sera pas content : il dira “non exhaustive pattern matching”.

Le menu

Pattern matching

En plus des types sommes, on peut faire du pattern matching sur n'importe quel type :

```
let string_to_color name =
  match name with
  | "red" -> Red
  | "green" -> Green
  | "blue" -> Blue
  | _ -> failwith "unknown color"
```

Il y a un soucis dans ce code, pouvez-vous le voir ?

Pattern matching

En plus des types sommes, on peut faire du pattern matching sur n'importe quel type :

```
let string_to_color name =
  match name with
  | "red" -> Red
  | "green" -> Green
  | "blue" -> Blue
  | _ -> failwith "unknown color"
```

Il y a un soucis dans ce code, pouvez-vous le voir ? Le `failwith` arrête tous le programme si la couleur n'est pas bonne...

Type somme : `option`

Quand on veut renvoyer un élément ou l'absence d'élément (si on ne peut pas le calculer), on utilise le type `option` :

```
type 'a option = Some of 'a | None
```

(le type `'a` est un type polymorphique, on y reviendra plus tard.)

Type somme : option

Quand on veut renvoyer un élément ou l'absence d'élément (si on ne peut pas le calculer), on utilise le type option :

```
type 'a option = Some of 'a | None
```

(le type 'a est un type polymorphique, on y reviendra plus tard.)

```
let string_to_color name =
  match name with
  | "red" -> Some Red
  | "green" -> Some Green
  | "blue" -> Some Blue
  | _ -> None
```

Le type de cette fonction est `string_to_color: string -> color option`.
(le type 'a est substitué par color dans la définition de option.)

Le menu

Les listes

- Les listes sont utilisées extensivement dans un programme fonctionnelle (au même titre que les tableaux dans un programme impératif).
- Elles méritent donc un peu de sucre syntaxique de la part du compilateur pour rendre leurs usages faciles.
- La page de doc : http://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html.

```
(* Liste vide *)
let empty = []
(* Liste avec trois elements *)
let l = [4; 2; 0]
(* Ajout d'un element en tete d'une liste. *)
cons x l      →   x :: l
(* Concatener deux listes. *)
append l1 l2  →   l1@l2
```

Le menu

- Types algébriques
- Conclusion

Conclusion

Les types algébriques sont au cœur de la modélisation dans le paradigme fonctionnel :

- Tuples
- Enregistrement
- Type somme
- Liste