Algorithmique et Approche Fonctionnelle de la Programmation

Cours 4 : Types structurés

28 septembre 2009

Louis Mandel
Université Paris-Sud 11
louis.mandel@lri.fr

D'après des transparents de Sylvain Conchon

Représentation de données structurées

- Les types de base (`int`, `float`, `bool`, `char`, `string`, `unit`) sont insuffisants pour représenter des données structurées (dates, matrices, ...)
  - ▷ des codages existent, mais ils ne sont pas naturels (et parfois aussi très inefficaces)
- Dans ce cours, nous allons étudier trois structures de données (et leurs opérations associées)
  - ▷ les produits (n-uplets)
  - ▷ les produits nommés (enregistrements)
  - ▷ les sommes

Les types produits

Le produit cartésien

$$
\begin{array}{rcl}
\textit{declaration} & ::= & \ldots \\
& | & \texttt{type}\ \textit{ident}\ \texttt{=}\ \textit{type-expression}\ \{\texttt{*}\ \textit{type-expression}\}^{+}\ [\texttt{;;}] \\
\textit{expression} & ::= & \ldots \\
& | & \textit{expression}\ \{\texttt{,}\ \textit{expression}\}^{+} \\
\textit{valeur} & ::= & \ldots \\
& | & \textit{valeur}\ \{\texttt{,}\ \textit{valeur}\}^{+}
\end{array}
$$

```
# type pair = int * int;;
type pair = int * int
# let a = 1, 2;;
val a : int * int = (1, 2)
```

- Il n'est pas nécessaire de déclarer les types produits

Construire des n-uplets

- Les éléments peuvent être de types différents

```
# let _ = (2.5, "coucou");;
- : float * string = (2.5, "coucou")
```

- Il peut y avoir un nombre arbitraire de composantes

```
# let _ = ("aaaa" ^ "bbb" ^ "cc", true || false, 1e10);;
- : string * bool * float = ("aaaabbbcc", true, 10000000000.)
```

- Les parenthèses sont optionnelles

```
# let _ = 1 + 3, not true;;
- : int * bool = (4, false)
```

- Les n-uplets peuvent être imbriqués

```
# let _ = ('a', (1.2, 1), true);;
- : char * (float * int) * bool = ('a', (1.2, 1), true)
```

Ne pas confondre

- Les expressions suivantes sont différentes
  - `(1, 2, 3)`
  - `(1, (2, 3))`
  - `((1, 2), 3)`
- Elles sont respectivement de type
  - `int * int * int`
  - `int * (int * int)`
  - `(int * int) * int`

Typage

- Type de l'expression $e_1\,,\ e_2\,,\ \ldots,\ e_n$:
  - si l'expression e_1 est de type t_1
  - si l'expression e_2 est de type t_2
  - $\ldots$
  - si l'expression e_n est de type t_n
  - alors $e_1\,,\ e_2\,,\ \ldots,\ e_n$ est de type $t_1\ *\ t_2\ *\ \ldots\ *\ t_n$
  - la règle de typage des n-uplet est :

$$\text{si}\quad H \vdash e_1 : t_1 \quad\text{et}\quad H \vdash e_2 : t_2 \quad\text{et}\quad \ldots \quad\text{et}\quad H \vdash e_n : t_n$$

$$\text{alors}\quad H \vdash e_1\,,\ e_2\,,\ \ldots,\ e_n : t_1\ *\ t_2\ *\ \ldots\ *\ t_n$$

Accès aux éléments d'un n-uplet

- Les fonctions `fst` et `snd` permettent d'accéder respectivement d'accéder à la première et la deuxième composante d'une paire

```
# let p = ((1.5, 1), 'a');;
val p : (float * int) * char = ((1.5, 1), 'a')
# let _ = fst p;;
- : float * int = (1.5, 1)
# let _ = snd p;;
- : char = 'a'
```

Accès aux éléments d'un n-uplet

- L'accès aux éléments d'un n-uplet peut se faire par filtrage

```
# let v = (1, 2, 3);;
val v : int * int * int = (1, 2, 3)
# let _ = match v with
          | x, y, z -> x + y + z;;
- : int = 6
# let x, y, z = v;;
val x : int = 1
val y : int = 2
val z : int = 3
# let (a, (b, c)) = ('\n', ("hello", "bye"));;
val a : char = '\n'
val b : string = "hello"
val c : string = "bye"
```

n-uplets : valeurs de première classe

- Les n-uplets peuvent être passés en arguments aux fonctions

```
# let f (x, y, z) = x + y * (int_of_float z);;
val f : int * int * float -> int = <fun>
# let _ = f (1, 2, 3.5);;
- : int = 7
```

- Les n-uplets peuvent être des résultats des fonctions

```
# let rec division n m =
    if n < m then (0, m)
    else
      let (q, r) = division (n - m) m in
      (q + 1, r)
  ;;
val division : int -> int -> int * int = <fun>
# let _ = division 5 3;;
- : int * int = (1, 3)
```

Exemple

- Calcul efficace de la fonction de Fibonacci

```
# let fibonacci n =
  (* fonction intermédiaire retournant (fib n, fib (n+1)) *)
    let rec fib_rapide n =
      if n = 0 then (0, 1)
      else
        let x, y = fib_rapide (n - 1) in
        (y, x + y)
    in
    fst (fib_rapide n)
  ;;
val fibonacci : int -> int = <fun>
# let _ = fibonacci 15;;
- : int = 610
```

Types produits et polymorphisme

```
# let first (x, y) = x;;
val first : 'a * 'b -> 'a = <fun>
# let _ = first  (1, 2);;
- : int = 1
# let _ = first ("aaa", "bbb");;
- : string = "aaa"

# let second (x, y) = y;;
val second : 'a * 'b -> 'b = <fun>
# let _ = second  (1, 2);;
- : int = 2
# let _ = second ("aaa", "bbb");;
- : string = "bbb"
```

Inconvénients des n-uplets

- Les objets représentés par des n-uplets ne sont pas identifiés de manière unique. Le système de type ne peut donc pas être utilisé pour garantir de « bonnes » propriétés
- Exemple : on veut représenter des nombres complexe par des paires `(t, i)` de type `float * float` où `r` et `i` sont respectivement la partie réelle et la partie imaginaire du complexe.
  - ▷ malheureusement, ce type peut tout aussi bien représenter des nombres complexes en notation polaire, ou des intervalles...
  - ▷ comment alors garantir que la fonction suivante est bien utilisée pour ajouter des complexes ?

```
# let add (r1, i1) (r2, i2) = (r1 +. r2, i1 +. i2);;
val add : float * float -> float * float -> float * float = <fun>
```

Inconvénients des n-uplets

- Les n-uplets avec un grand nombre de composantes deviennent très vite inutilisables en pratique
- Exemple : une fiche d'un fichier de personnes (nom, prénom, adresse, date de naissance, téléphone fixe, téléphone portable, etc.)

```
let v =
  ("Durand","Jacques",
   ("2 rue J.Monod", "Orsay Cedex", 91893),
   (10,03,1967), "0130452637","0645362738", ...)
```

  - la consultation des informations devient vite pénible
  - plusieurs éléments du n-uplet peuvent avoir le même type (ex. nom et prénom) et il est facile de les confondre (sans que le système de type puisse nous aider)

Produit nommés

Les enregistrements

- Le produit nommé permet de définir les enregistrements
  - ▷ n-uplets dont les éléments (champs) ont chacun un nom *distinct*
- En OCaml, chaque produit nommé (ou type enregistrement) possède un nom donné par l'utilisateur
- Les enregistrements sont comme les structures de C

Types enregistrements

$$
\begin{array}{rcl}
\textit{declaration} & ::= & \ldots \\
& | & \texttt{type}\ \textit{ident}\ \texttt{=}\ \texttt{\{}\ \{\textit{label}\ \texttt{:}\ \textit{type-expression}\ \texttt{;}\}^{+}\ \texttt{\}}\ [\texttt{;;}] \\
\textit{expression} & ::= & \ldots \\
& | & \texttt{\{}\ \{\textit{label}\ \texttt{=}\ \textit{expression}\ \texttt{;}\}^{+}\ \texttt{\}}
\end{array}
$$

- Il faut déclarer le type enregistrement

```
# type complexe = { re: float; im: float; };;
type complexe = { re : float; im : float; }
```

- Allocation et initialisation simultanées

```
# let cpx = { re = 1.0; im = -1.0; };;
val cpx : complexe = {re = 1.; im = -1.}
```

Accès aux éléments d'un enregistrement

$$\begin{array}{rcl} \textit{expression} & ::= & \ldots \\ & \mid & \textit{expression} \texttt{ . } \textit{nom-de-champ} \end{array}$$

- L'accès aux champs d'un enregistrement se fait à l'aide de la notation « . »

```
# let _ = cpx.re;;
- : float = 1.
# let _ = { re = 1.3; im = 0.9; }.im;;
- : float = 0.9
```

Accès aux éléments d'un enregistrement

- L'accès aux champs peut se faire aussi par filtrage

```
# let _ = match cpx with
          | { re = a; im = b; } -> a ;;
- : float = 1.
# let { im = b } = cpx;;
val b : float = -1.
```

  - ▷ et en profondeur

```
# type t1 = { ch1: complexe; ch2: int * string; };;
type t1 = { ch1 : complexe; ch2 : int * string; }
# let { ch1 = { re = x; im = y;}; ch2 = (_, z); } =
    { ch1 = cpx; ch2 = (1, "f") };;
val x : float = 1.
val y : float = -1.
val z : string = "f"
```

L'ordre des champs n'a pas d'importance

- Les définitions de type suivantes sont équivalentes

```
# type t = { a : int; b : float * char; c : string };;
```

```
# type t = { b : float * char; a : int; c : string };;
```

- De même, ces valeurs sont égales

```
# let _ =
    { a = 1; b = (1.1, 'g'); c = ""; } =
    { c = ""; b = (1.1, 'g'); a = 1; };;
```

```
- : bool = true
```

- Le filtrage est aussi insensible à l'ordre des champs

```
# let { c = x; b = y; } =
    { a = 1; b = (1.1, 'g'); c = ""; };;
```

```
val x : string = ""
val y : float * char = (1.1, 'g')
```

Structurer l'information

- Le mélange des n-uplets et des enregistrements permet de définir des objets complexes

```
# type adresse = { rue : string; ville : string; cp : int};;
```

```
# type fiche = {
    nom : string;
    prenom : string ;
    adresse : adresse;
    date_naissance : int * int * int;
    tel_fixe : string;
    portable : string
  };;
```

Création de valeurs

- On peut créer un nouvel enregistrement en utilisant le contenu des champs d'un autre enregistrement

```
# let v1 = { a = 1; b = (1.0, 'r'); c = "bonjour"};;
val v1 : t = {b = (1., 'r'); a = 1; c = "bonjour"}
# let v2 = { a = v1.a; b = v1.b; c = "hello"};;
val v2 : t = {b = (1., 'r'); a = 1; c = "hello"}
```

- Le raccourci suivant permet d'arriver au même résultat

```
# let v3 = { v1 with c = "hello" };;
val v3 : t = {b = (1., 'r'); a = 1; c = "hello"}
```

Création de valeurs

```
# type paire = { gauche: int; droite: int; };;
type paire = { gauche : int; droite : int; }
# type triplet =
    { gauche: int; milieu: int; droite: int; };;
type triplet = { gauche : int; milieu : int; droite : int; }

# let _ = { gauche = 1; milieu = 1; droite = 1; };;
- : triplet = {gauche = 1; milieu = 1; droite = 1}
```

- Il ne peut pas y avoir plusieurs types enregistrements avec les mêmes noms de champs

```
# let _ = { gauche = 1; droite = 1; };;
Error: Some record field labels are undefined: milieu
```

Enregistrements : valeurs de première classe

- Une fonction prenant un enregistrement en argument

```
# let f v = v.a;;
val f : t -> int = <fun>
# let _ = f { a = 1; b = (0.0, 'e'); c = ""; };;
- : int = 1
```

- Les enregistrements peuvent aussi être retournés en résultat

```
# let f { a = x } v = { v with a = x + v.a; };;
val f : t -> t -> t = <fun>
# let _ =
    let v = { a = 1; b = (0.0, 'e'); c = ""; } in
    f v v;;
- : t = {b = (0., 'e'); a = 2; c = ""}
```

Types enregistrements et polymorphisme

Il est possible de déclarer des types polymorphes

```
# type 'a valeur_nommee =
    { nom : string;
      valeur : 'a };;
type 'a valeur_nommee = { nom : string; valeur : 'a; }
```

```
# let _ = { nom = "v1"; valeur = 1; };;
- : int valeur_nommee = {nom = "v1"; valeur = 1}
```

```
# let _ = { nom = "v2"; valeur = true; };;
- : bool valeur_nommee = {nom = "v2"; valeur = true}
```

Types sommes

Types sommes

$$\begin{array}{lcl} \textit{declaration} & ::= & \ldots \\ & \mid & \texttt{type}\ \textit{ident}\ \texttt{=}\ \{\texttt{|}\ \textit{u-ident}\ [\texttt{of}\ \textit{type-expression}]\}^{+}\ [\texttt{;;}] \end{array}$$

- Les types sommes permettent de modéliser les domaines finis

```
# type saison = Printemps | Ete | Automne | Hiver;;
type saison = Printemps | Ete | Automne | Hiver
```

  - Il y a exactement 4 valeurs de type `saison`

```
# let e = Ete;;
val e : saison = Ete
```

  - Les identificateurs `Printemps`, `Ete`, `Automne` et `Hiver` sont des constructeurs (les majuscules sont obligatoires pour définir des constructeurs)

Types sommes

- Les types sommes permettent de faire l'union de plusieurs types

```
# type num =
    | Entier of int
    | Flottant of float
  ;;
type num = Entier of int | Flottant of float
```

  - les valeurs de types `num` sont des constructeurs associés à des valeurs

```
# let v1 = Entier 1;;
val v1 : num = Entier 1
# let v2 = Flottant 10.3;;
val v2 : num = Flottant 10.3
```

Types sommes

Les types `unit` et `list` sont des types sommes

```
# type my_unit = Unit;;
type my_unit = Unit
# type 'a my_list =
    | Nil
    | Cons of 'a * 'a my_list;;
type 'a my_list = Nil | Cons of 'a * 'a my_list
```

Accès aux éléments

Pour utiliser une valeur de type somme, on fait des traitement par cas

```
# let print_saison s =
    match s with
    | Printemps -> print_string "printemps\n"
    | Ete -> print_string "été\n"
    | Automne -> print_string "automne\n"
    | Hiver -> print_string "hiver\n";;
val print_saison : saison -> unit = <fun>
```

```
# let _ = print_saison Automne;;
automne
- : unit = ()
```

Accès aux éléments

```
# let ajoute x y =
    match (x,y) with
    | (Entier(m) , Entier(n)) ->
        Entier(m + n)
    | (Entier(m) , Flottant(n)) ->
        Flottant((float_of_int m) +. n)
    | (Flottant(m) , Entier(n)) ->
        Flottant(m +. (float_of_int n))
    | (Flottant(m) , Flottant(n)) ->
        Flottant(m +. n);;
val ajoute : num -> num -> num = <fun>
# let _ = ajoute (Flottant 3.2) (Entier 1);;
- : num = Flottant 4.2
```

- Le compilateur vérifie que le filtrage est complet

Types sommes

```
# type formule =
    | Vrai
    | Faux
    | Conjonction of formule * formule
  ;;
type formule = Vrai | Faux | Conjonction of formule * formule
# let _ = Vrai;;
- : formule = Vrai
# let _ = Conjonction (Vrai, Faux);;
- : formule = Conjonction (Vrai, Faux)
```

Filtrage

```
# let rec evalue e =
    match e with
    | Vrai -> true
    | Faux -> false
    | Conjonction (e1, e2) -> evalue e1 && evalue e2
  ;;
val evalue : formule -> bool = <fun>
# let _ =
    evalue (Conjonction (Faux, Conjonction (Vrai, Faux)));;
- : bool = false
```

Filtrage

les motifs de filtrage peuvent être imbriqués

```
# let rec evalue e =
    match e with
    | Vrai -> true
    | Faux -> false
    | Conjonction (Faux, e2) -> false
    | Conjonction (e1, Faux) -> false
    | Conjonction (e1, e2) -> evalue e1 && evalue e2
  ;;
val evalue : formule -> bool = <fun>
```

Filtrage

les motifs de filtrage peuvent être omis ou regroupés

```
# let rec evalue e =
    match e with
    | Vrai -> true
    | Faux -> false
    | Conjonction (Faux, _) | Conjonction (_, Faux) -> false
    | Conjonction (e1, e2) -> evalue e1 && evalue e2
  ;;
val evalue : formule -> bool = <fun>
```

Type option

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

- Définition de fonctions partielles

```
# let div x y =
    if y = 0 then None
    else Some (x/y)
  ;;
val div : int -> int -> int option = <fun>
```