Algorithmique fonctionnelle - Types de données

4 février 2013

Rappel

Types de base

OCaml propose une liste de types de base, dits primitifs :

- Numériques : `float, int`
- Alphanumériques : `char, string`
- Booléens : `bool`

Rappel

Tuples

Il est possible de composer les types entre eux, en créant des produits cartésiens ou des tuples :

- let x = ("toto", 12) ; ; : string * int
- let y = (12.5, '8', true) ; ; : float * char * bool
- let z = (("Zaphod", "Beeblebrox"), 42) ; ; : (string * string) * int

Produit cartésien

Accesseurs

Les fonctions **`fst`** et **`snd`** permettent d'accéder aux éléments du produit.

Exemple

```
let x = ("toto", 12);;
(fst x);; (* "toto" *)
(snd x);; (* 12 *)
(fst z);; ("Zaphod", "Beeblebrox")
(fst y);;
Error : This expression has type float * char * bool
but an expression was expected of type 'a * 'b
```

Tuples

Définition

Un tuple est une généralisation du produit cartésien. Pour accéder à ses éléments, on le "mappe" avec un tuple de variables.

Exemple

```
let y = (12.5, '8', true) ; ; : float * char * bool
let (a,b,c) = y ; ;
a ; ;
```

Tuples

Dans les fonctions

Un tuple peut être un paramètre d'entrée ou un retour de fonction.

Exemple

```
let addcmplx (r1,i1) (r2,i2) = (r1 +. r2, i1 +. i2);;

let rec quotrestrec a b q =
   if a < b then
      (q,a)
   else
      quotrestrec (a - b) b (q+1);;

let quotrest a b = quotrestrec a b 0;;
```

Tuples

Inconvénients

Il existe plusieurs inconvénients à utiliser des tuples pour structurer des données :

- Difficilement manipulable à multiples composantes
  (ex : pour un individu,
  ((nom,prenom),(rue,cp,ville),(journaiss,moisnaiss,anneenaiss)))
- Un tuple ne peut définir un élément de manière unique :
  un couple d'entier peut définir le quotient et le reste d'une division,
  mais tout aussi bien un intervalle dans $\mathbb{N}$
- Nécessité de structurer les données afin de sécuriser un programme et d'apporter une meilleure lecture

Types structurés

Présentation

Définir des tuples pour structurer des données en nommant leurs éléments :

- **`type`** nom **`=`** { nomchp1 : type1 **`;`** nomchp2 : type2 **`;`** ...}
- Le type défini est une **structure**
- Les éléments sont les **champs**. Ils sont définis par leur **étiquette**
- Pas d'ordre sur les champs
- Les types des champs sont des types primitifs ou d'autres structures
- Accession : variable**`.`**nomchamp

Construire un type

Exemple : individu

```
type adresse = {rue : string ; cp : int ; ville : string} ; ;
type datenaiss = {jour : int ; mois : int ; annee : int} ; ;
type individu = {nom : string ; prenom : string ; adr : adresse ; naiss : datenaiss}

let mari = {nom="Toto" ; prenom="Jean-Michel" ;
        adr={rue="12 rue Tabaga" ; cp=17342 ; ville="Moulinot-les-bains"} ;
        naiss={jour=6 ; mois=11 ; annee=1984}} ; ;
mari.adr.rue ; ;
```

Construire un type

Création à partir de...

```
let femme = {mari with prenom="Augustine"};;
let enfant = {femme with prenom="Titou";
    naiss={jour=7; mois=femme.naiss.mois;annee=2006}};;

femme.prenom;;
enfant.naiss;;
```

Nombres complexes

Définition

Un nombre complexe est défini par une partie réelle et une partie imaginaire.

Exemple

```
type complexe = {re : float ; im : float} ; ;
let x = {re=3. ; im=7.2} ; ;
x.im ; ;
```

Nombre complexe

Définition

Utilisation dans une fonction

- En paramètre :

```
let module x = sqrt(x.re**2. +. x.im**2.) ;;
```

- En résultat :

```
let creercmplx x y = {re=x ; im=y} ;;
let pluscmplx x y = {re=(x.re +. y.re) ; im=(x.im +. y.im)} ;;
```

Types sommes

Définition

Les types sommes permettent de regrouper, au sein d'un même type, différents types.

- **Union disjointe** d'ensembles de valeurs
- Définit un ensemble de **constructeurs** pour chaque élément

Types sommes

Définition

type nom =
Constructeur$_1$ [**of** type$_1$]
| ...
| **C**onstructeur$_n$ [**of** type$_n$]

Types sommes

Exemple : Feux tricolores

```
type couleurFeu = Rouge | Orange | Vert ;;
type feu = {id : string ; couleur : couleurFeu} ;;
let coul = Rouge ;;

let feu1 = {id="feuA" ; couleur=coul} ;;
feu1.couleur=Vert ;; (* bool = false *)
let feu1 = {feu1 with couleur=Vert} ;;
feu1.couleur = Bleu ;;
```

Types sommes

Définition

Un conctructeur permet d'initialiser une valeur de l'ensemble.

- Sans paramètre (cf. exemple précédent)
- Avec paramètre :

```
type mois = Mois of int ;;
let m = Mois(6) ;;
m + 2 ;;
match m with Mois a -> a + 2 ;;
let m = Mois(42) ;;
```

Types sommes

Définition

Il est possible de paramétrer le constructeur afin de fournir un sous-ensemble de définition.

Exemple : Mois

```
type mois = Mois of int ;;

let creerMois = function m ->
if (m>=1 && m <= 12) then Mois m
else failwith(" creerMois : mois invalide ");;

let getValMois = function Mois m -> m ;;
```

Types sommes

Exemple : Cartes à jouer

```
type valeur = Roi | Dame | Valet | Nombre of int ;;
let creerVal = function v -> match v with
Roi | Dame | Valet -> v
| Nombre c -> if (c>=1 && c <= 10) then v
        else failwith(" creerVal : valeur invalide ")
|_ -> failwith(" creerVal : valeur invalide ") ;;

type couleur = Pique | Coeur | Carreau | Trefle ;;
type carte = {vale : valeur ; coul : couleur} ;;
let creerCarte v c = {vale=v ; coul=c} ;;

let carte1 = (creerCarte (creerVal Roi) Trefle) ;;
let carte2 = (creerCarte (creerVal (Nombre 1)) Pique) ;;
let carte3 = (creerCarte (creerVal (Nombre 12)) Pique) ;;
let carte4 = (creerCarte (creerVal 7) Coeur) ;;
```

Types sommes

Exemple : Cartes à jouer

```
(* précondition : c1, c2 cartes de valeurs différentes *)
(* postcondition : vrai si la valeur de c1 est supérieure à celle de c2 *)
let plusgrand c1 c2 = match (c1.vale,c2.vale) with
(Nombre 1,_) -> true
| (Roi,Nombre 1) -> false
| (Roi,_) -> true
| (Dame,Nombre 1) -> false
| (Dame,Roi) -> false
| (Dame,_) -> true
| (Valet,Nombre 1) -> false
| (Valet,Nombre n) -> true
| (Valet,_) -> false
| (Nombre n, Nombre m) -> n > m
| _ -> false;;
```

Types sommes

Exemple : Cartes à jouer

```
let valToInt v = match v with
Nombre 1 -> 14
| Roi -> 13
| Dame -> 12
| Valet -> 11
| Nombre n -> n ; ;

(* précondition : c1, c2 cartes de valeurs différentes *)
(* postcondition : vrai si la valeur de c1 est supérieure à celle de c2 *)
let plusgrand c1 c2 = valToInt(c1.vale) > valToInt(c2.vale) ; ;

plusgrand carte1 carte2 ; ;
```

Types récursifs

Définition

Un type peut être défini par lui-même, on parle alors de **type récursif**.
Par exemple, une formule mathématique peut se représenter comme composé de formules et d'opérations entre ces formules.

Types récursifs

Exemple : Arbre de calcul

Afin de représenter une formule de calcul (d'entiers), définissons :

- 4 opérations : plus, moins, fois, div
- Des constantes numériques, entières
- Des variables

Types récursifs

Exemple : Arbre de calcul

On peut alors définir le type selon :

```
type formule = Plus of formule * formule
| Moins of formule * formule
| Fois of formule * formule
| Div of formule * formule
| Const of int
| Var of string ;;
```

Types récursifs

Exemple : Arbre de calcul

La formule

$$2 * (x - 3) + (y/5 - 1)$$

peut se représenter par :

```
let formu = Plus (
      Fois (Const 2, Moins (Var "x", Const 3)),
        Moins (Div (Var "y", Const 5), Const 1)) ; ;
```