Types Sommes avec constructeurs

AP1

Catherine Dubois

Les types de base et les constructeurs de types (list, $\rightarrow$, *) ne sont pas suffisants ou mal commodes pour

- définir des types énumérés
- construire de nouveaux ensembles de valeurs
- représenter des unions de valeurs de types différents
- distinguer des types semblables

$\Rightarrow$ solution : **types sommes avec constructeurs** (ou encore types concrets, types algébriques)

Un type somme est nommé explicitement (déclaration):

```
type nom_du_nv_type =  définition du nv type;;
```

La définition indique les moyens de construire les valeurs de ce type : notion de **constructeurs**
Un constructeur est :

- soit une constante (c'est un nom)
- soit un constructeur non constant qui est appliqué à des valeurs pour obtenir une valeur du nouveau type

Syntaxe de la déclaration d'un type somme :

```
type  nom_du_nv_type = C_1 | ... | C_p   (* constantes *)
|  C'_1 of T_1 | ...  | C'_k of T_k;;
```

avec $p + k \geqslant 1$, les T_i sont des types
Les constantes et les constructeurs doivent obligatoirement commencer par une majuscule

• Type somme avec constantes seulement : Type énuméré

```
#type direction = Nord | Sud | Est | Ouest;;
 une direction est Nord, Sud, Est ou Ouest
 direction = {Nord, Sud, Est, Ouest}

#Est;;
- : direction = Est

#succ(Est);;
This expression has type direction but is here used
with type int

#Nord = Sud;;
- : bool = false
```

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

#type tige = A | B | C;;

#type singleton = One;;
```

Le type prédéfini `bool` est un type énuméré

```
type bool = true | false;;
```

```
#let tourne d = match d with
                  Nord -> Ouest
                | Ouest -> Sud
                | Sud -> Est
                | Est -> Nord;;
tourne : direction -> direction =>fun>
```

Une constante peut être un filtre

- **Type avec un seul constructeur (non constant)**

```
#type longueur = L of int;;
 une longueur est en fait un entier muni d'une étiquette

#type surface = S of int;;

#let aire_rect ((L l1), (L l2)) = S (l1 * l2);;  filtrage
val aire_rect : longueur * longueur -> surface = <fun>

#(aire_rect ((L 2), (L 3)) ) + 5;;
This expression has type surface but is here used
with type int
```

Le but est ici d'utiliser le typage pour des vérifications de cohérence (mais écriture plus lourde).

Attention : `type lg == int;;`
ne définit pas un nouveau type mais renomme un type

- **Formes géométriques**

On veut définir le type `forme` des figures géométriques qui sont

- soit des points,
- soit des cercles,
- soit des rectangles

Un cercle est caractérisé par son rayon

Un rectangle est caractérisé par sa longueur et sa largueur.

```
type forme = Point | Cercle of float
                   | Rectangle of float*float;;
```

L'ensemble des valeurs de type `forme` est
{Point} ∪ {Cercle r | r est un flottant}
∪ { Rectangle (l, m) | l, m flottants}

```
let F1 = Point;;
val F1 : forme = Point

let F2 = Cercle 0.5;;
val F2 : forme = Cercle 0.5

let F3 = Rectangle (3.2, 5.6);;
val F3 : forme =  Rectangle (3.2, 5.6)

let l_geom = [Point; F2; F3; F1 ];;
val l_geom : forme list =
[Point; Cercle 0.5; Rectangle (3.2,5.6); Point];;

type boîte =   Cube of float
  | Parallélépipède of float * float * float;;
```

que des constructeurs non constants

Typage : Une expression de la forme `Constructeur_i e` est de type `nom_du_type` si `e` est du type `type_i`.

Par exemple `Hiver 3` erreur de typage
`Parallélépipède (3.0, 2.1, 5.6)` a le type boîte.

Évaluation : Une expression de la forme `Constructeur_i e` a pour valeur `Constructeur_i v` si l'expression `e` a pour valeur `v`.

```
Cercle (5.2 *. 2.0);;
- : forme = Cercle 10.4
```

- *Une forme géométrique est-elle un point?*

```
let est_un_point f = (f = Point);;
val est_un_point : forme -> bool

est_un_point F1;;
- : bool = true

est_un_point F2;;
- : bool = false
```

- *Une forme géométrique est-elle un rectangle?*

Ici une simple comparaison ne suffit pas !

⇒ analyser le constructeur du paramètre de type `forme`

⇒ par filtrage, comme d'habitude

```
let est_un_rectangle f = match f with
    Rectangle _ -> true
    | _ -> false;;
val est_un_rectangle : forme -> bool

est_un_rectangle F1;;
- : bool = false

est_un_rectangle F3;;
- : bool = true
```

```
let est_un_carré f = match f with
| Rectangle (lo, la) -> lo=la
| _ -> false;;

let aire  = let pi = 3.14 in function f ->
  match f with
| Point -> 0.0
| Cercle r -> pi *. r *. r
| Rectangle (lo, la) -> lo *. la;;
```

• *Modélisation d'un jeu de tarot*
cartes ordinaires, cartes d'atout et l'excuse.

```
type couleur = Trèfle | Pique | Coeur | Carreau;;

type carte_ordi =
| Roi of couleur
| Dame of couleur
| Cavalier of couleur
| Valet of couleur
| Chiffre of int*couleur;;

type carte = Excuse | Atout of int
                    | Carte_couleur of carte_ordi;;

let main =
[Excuse; Atout 1; Atout 21; Carte_couleur (Roi Trèfle)];;
main : carte list = ...
```

Bilan sur les filtres

Aux filtres précédemment présentés, on a ajouté des filtres de la forme `Constructeur_i` si ce constructeur est un constructeur constant ou `Constructeur_i f` si le constructeur `Constructeur_i` est non constant et `f` est un filtre du type `type_i`.

En général

les fonctions définies sur un type somme opèrent souvent par filtrage

On se laisse **guider** par la définition du type :

$\Rightarrow$ squelette du filtrage à écrire (au moins un cas par constructeur + cas particuliers)

- **Autres exemples**

```
#type réponse = Aucune  | Solution of int;;

#let division (a, b) =
  if b=0 then Aucune else Solution (a/b);;
val division : int*int -> réponse  fonction totale

#let division2 (a, b) =
  if b=0 then failwith "impossible" else a/b;;
val division2 : int*int -> int
 fonction partielle : non définie si y=0

# division (5, 0);;
- : réponse = Aucune

# division2 (5, 0);;
Uncaught exception: Failure("impossible")

#type monnaie = Euros of int | Dollars of int;;
```

- **Les types sommes peuvent être paramétrés (polymorphes)**

```
#type 'a réponse = Aucune  | Solution of 'a;;

#let division (a, b) =
  if b=0 then Aucune else Solution (a/b);;
val division : int*int -> int réponse = <fun>

#let divisionf (a, b) =
  if b=0.0 then Aucune else Solution (a/.b);;
val divisionf : float*float -> float réponse = <fun>

#type ('a, 'b) réponse = Aucune  | Solution of 'a
                       | AutreSolution of 'b;;

#let f x = if x = 0 then Aucune
           else if x<0 then Solution true
                else AutreSolution "un texte";;
val f : int -> (bool, string) réponse = <fun>
```

• Les types récursifs

Les types sommes sont par défaut récursifs si le nom du type apparait dans la définition

```
#type oignon =  Trognon | Pelure of oignon;;

#type nat = Zero | S of nat;;

#Zero;;                          S( S Zero);;
- : nat = Zero                   - : nat = S (S Zero)
```

Ceci correspond à une définition inductive ⇒ principe d'induction structurelle
Soit P(n) avec n de type `nat`
Démontrer P(n) pour tout n de type `nat` *c'est :*
démontrer P(`Zero`*) et démontrer P(S n) en supposant P(n).*

```
let rec plus_nat (m,n) = match m with
                          Zero -> n
                        | (S p) -> S (plus_nat (p, n))
val plus_nat : nat * nat -> nat = <fun>

plus_nat ((S (S Zero)), (S (S Zero)));;
- : nat = S (S (S (S Zero)))

plus_nat  S (S Zero), S (S Zero) =
S (plus_nat S Zero, S (S Zero)) =
S (S (plus_nat Zero,  S (S Zero))) =
S (S  (S (S Zero)))
```

Le plus souvent : type récursif $\Rightarrow$ fonctions définies récursivement par filtrage

Deux grandes applications :

- définir les structures de données classiques de l'algorithmique (les listes, les files, les arbres ...)
- définir des syntaxes abstraites (des expressions, des termes, des programmes ...)

•Listes

Le type des listes d'entiers :

```
type listint = Nil | Cons of int*listint;;
Cons (1, Cons (2, Nil));;
- : listint = Cons (1, Cons (2, Nil))
```

Le type des listes polymorphes :

```
type 'a mon_list = Nil | Cons of 'a * 'a mon_list;;

Cons (1, Cons (2, Nil));;
- : int mon_list = Cons (1, Cons (2, Nil))   1::2::[]
```

•Piles

Le type des piles :

```
type 'a pile = Pilevide | Push of 'a * 'a pile;;

let ma_pile = Push (1, Push (2, Pilevide));;
val ma_pile : int pile =  Push (1, Push (2, Pilevide))

let empiler (x, p) = Push (x,p);;
val empiler : 'a*'a pile -> 'a pile = <fun>

let dépiler pi = match pi with
                    Pilevide -> failwith "dépiler"
                  | Push (_,p) -> p;;
val dépiler : 'a pile -> 'a pile = <fun>

let rec hauteur pi = match pi with
                      Pilevide -> 0
                     | Push(_,p) -> 1 + (hauteur p);;
val hauteur : 'a pile -> int = <fun>
```

•La structure d'arbre

très utilisée en informatique pour :

- présenter un ensemble d'objets ou d'informations élémentaires en une structure hiérarchique
- faciliter l'accès ou la recherche de ces informations
- modéliser de nombreux problèmes

Complément indispensable de la liste
liste = organisation linéaire des données : tous les objets sont au même niveau

Exemples de modélisation : livre, fichiers d'un système d'exploitation, expression arithmétique, arbres de classification (instruments de musique), arbre de jeu entre adversaires ...

Représentation ensembliste, indentée, graphique

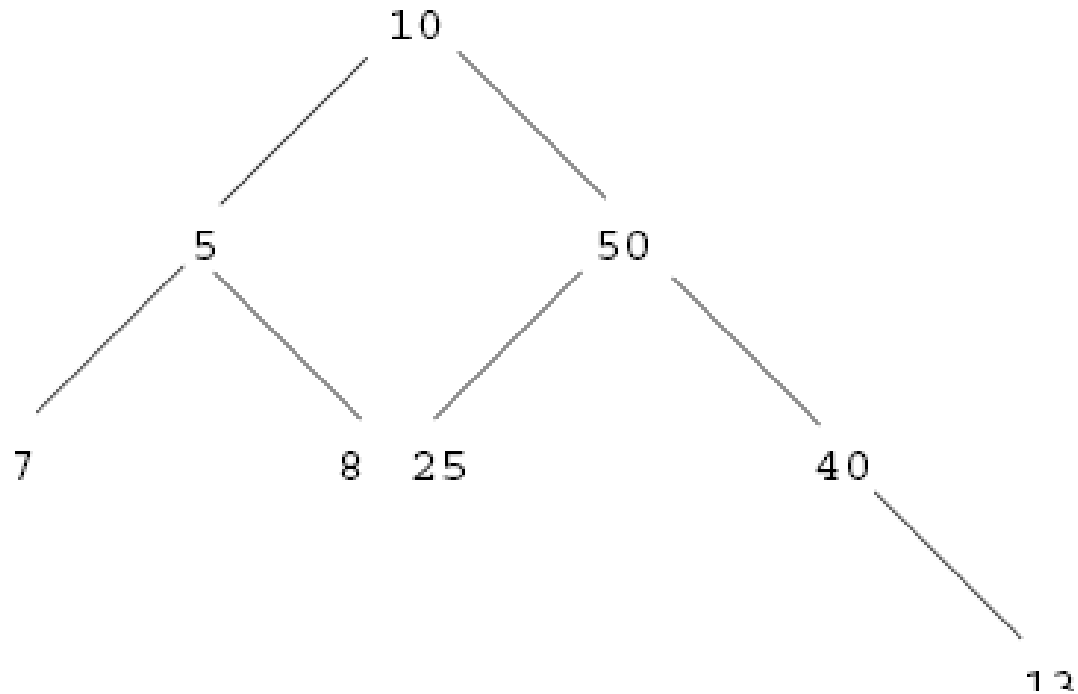
En info : arbres sont à l'envers (racine en haut, les branches tournées vers le bas, feuilles en bas !!!)

Un *arbre* est une structure de données hiérarchique qui est :

- soit vide
- soit un *nœud* contenant une donnée (encore appelée étiquette au nœud) et des sous-arbres éventuellement vides.

Un arbre non vide, ayant uniquement des sous-arbres vides, est appelé une *feuille*

On s'intéresse dans la suite aux arbres binaires où chaque arbre non vide possède deux sous-arbres éventuellement vides : l'un est appelé *sous-arbre gauche* et l'autre *sous-arbre droit*


10
5
50
7
8
25
40

Le type des arbres binaires d'entiers :

```
type arbre_binaire =
| Vide
| Noeud of int * arbre_binaire * arbre_binaire;;

let ex_arbre =
  Noeud(10, Noeud(5, Noeud(7,Vide,Vide),
                     Noeud(8,Vide,Vide)),
          Noeud(30, Vide,
                    Noeud(40,Vide,Vide)));;

let e1 = Noeud(30,Noeud(10,Vide,Vide),Noeud(40,Vide,Vide));

let e2 = Noeud("coucou",Vide,Noeud(2,Vide,Vide));;
 erreur de typage !!!! mélange de chaînes de caractères
et d'entiers
```

```
let est_feuille a = match a with
  Noeud(_, Vide, Vide) -> true
| _ -> false;;
val est_feuille :                                  = <fun>

let rec nbnoeuds a = match a with
  Vide -> 0
| Noeud(_,sag,sad) -> 1+ (nbnoeuds sag) + (nbnoeuds sad);;
val nbnoeuds :                                     = <fun>
```

```
nbnoeuds e1 = nbnoeuds Noeud(30,
                            Noeud(10, Vide, Vide),
                            Noeud(40, Vide, Vide)) =
1+ nbnoeuds (Noeud(10, Vide, Vide)) +
   nbnoeuds (Noeud(40, Vide, Vide)) =
1 + 1 + nbnoeuds Vide + nbnoeuds Vide +
   nbnoeuds (Noeud(40, Vide, Vide)) =
1 + 1 + 0 + 0 + 1 + nbnoeuds Vide + nbnoeuds Vide =
1 + 1 + 0 + 0 + 1 + 0 + 0
```

Exercice : sommer les valeurs des noeuds

Recherche d'un élément dans l'arbre :

```
let rec est_dans (e, a) = match a with
| Vide -> false
| Noeud (r, sag, sad) ->
   (r = e) || est_dans (e, sag) || est_dans (e, sad);;

type : ?
```

Plus généralement le type des arbres paramétré par le type des éléments : `arbre_bin` n'est plus un type mais un moyen de construire des types

```
type 'a arbre_bin =
| Vide
| Noeud of 'a * 'a arbre_bin * 'a arbre_bin;;

type 'a arbre_bin =
| Vide
| Noeud of 'a * 'a arbre_bin * 'a arbre_bin;;
```

```
Noeud (1, Vide, (Noeud (2, Vide, Vide)))
```
a le type int arbre_bin

```
Noeud ("ab",(Noeud ("AZERTY", Noeud
("bc",Vide,Vide),Vide)),Vide)
```
a le type string arbre_bin

Quels sont les nouveaux types de ces fonctions ?

```
let est_feuille a = match a with
  Noeud(_, Vide, Vide) -> true
| _ -> false;;
val est_feuille : 'a arbre_bin -> true = <fun>

let rec nbnoeuds a = match a with
  Vide -> 0
| Noeud(_,g,d) -> 1+ (nbnoeuds g) + (nbnoeuds d);;
val nbnoeuds : 'a arbre_bin -> int = <fun>

let rec est_dans (e, a) = match a with
| Vide -> false
| Noeud (r, sag, sad) ->
  (r = e) || est_dans (e, sag) || est_dans (e, sad);;
val est_dans : 'a * 'a arbre_bin -> bool = <fun>
```

Ajoutons un ordre sur les éléments rangés dans l'arbre et des contraintes de rangement

Par exemple, tous les éléments dans le sous arbre gauche sont plus petits que la racine, tous les éléments dans le sous arbre droit sont plus grands que la racine

⇒ arbre binaire de recherche (ABR)

Cette propriété permet d'améliorer la recherche d'un élément dans un arbre binaire de recherche.

```
let rec est_dans_abr (e, a) = match a with
| Vide -> false
| Noeud(r, sag, sad) ->
     if r = e then true
     else if (e < r) then est_dans_abr (e,sag)
          else est_dans_abr (e, sad);;
```

⇒ méthode dichotomique

•Expressions arithmétiques

Définissons le langage des expressions arithmétiques composées d'entiers, d'additions, de soustractions et de multiplications.

2 syntaxes en informatique

- syntaxe concrète SC (utilisée pour les entrées-sorties sous forme de chaînes de caractères, par exemple $2 + (3 * 5)$)
- syntaxe abstraite SA (représentation interne des objets : on ne retient que la structure)

expression en syntaxe concrète $\Longrightarrow$ expression en syntaxe abstraite $\Longrightarrow$ calcul (simplification par exemple) sur l'expression en SA

```
 (* syntaxe abstraite des expressions arithmétiques *)
type expr = Nb of int | Plus of expr*expr | Mult of expr*ex

 (* Exemples d'expressions *)
Nb 0;;
Mult (Nb 5, Nb 3);;
Mult (Plus (Nb 0, Nb 3), Mult (Nb 4, Nb 3));;
```

On dispose pour prouver des propriétés sur les expressions du principe d'induction structurelle

Évaluation des expressions

```
#(*interface évaluation
type : expr -> int
argument e
postcondition : calcule la valeur entière de l'expression
*)
let rec évaluation e = match e with
| Nb n -> n
| Plus (e1, e2) -> (évaluation e1)+(évaluation e2)
| Mult (e1, e2) -> (évaluation e1)*(évaluation e2);;
val évaluation : expr -> int = <fun>

#évaluation (Plus (Nb 1, Plus (Nb 2, Nb 5)));;
- : int = 8
```

Ajoutons des variables à ce langage

```
type expr =   Var of string | Nb of int
            | Plus of expr * expr | Mult of expr * expr;;
```

Exemples d'expressions

```
Var "x";;
Mult (Var "x", Nb 3);;
Mult (Plus (Var "x", Nb 3), Plus (Var "x", Nb 3));;
```

```
#let rec a_une_variable e = match e with
  Var _ -> true | Nb _ -> false
| Plus (e1, e2) -> a_une_variable e1 || a_une_variable e2
| Mult (e1, e2) -> a_une_variable e1 || a_une_variable e2;;
val a_une_variable : expr -> bool = <fun>

#a_une_variable (Plus (Nb 1, Plus (Nb 2, Nb 5)));;
- : bool = false

#a_une_variable (Plus (Var "x", Plus (Nb 2, Nb 5)));;
- : bool = true
```

Evaluation des expressions avec variables

L'évaluation de l'expression x+(2+5) ne peut se faire que l'on connaît la valeur de x

$\Rightarrow$ environnement (liste de couples *(nom de variable, valeur)*)

Exercice : Réécrire `évaluation` dans ce cadre

Dérivation d'expressions

```
let rec derive (x, e) = match e with
    Nb _ -> Nb 0
  | Var y -> if x=y then Nb 1 else Nb 0
  | Plus (e1, e2) -> Plus (derive (x, e1), derive (x, e2))
  | Mult (e1, e2) -> Plus( Mult(derive (x, e1), e2),
                           Mult(e1, derive (x, e2)));;
val derive : string * expr -> expr = <fun>

derive ("x", (Mult (Var "x", Var "y")));;
-:expr= Plus (Mult (Nb 1, Var "y"), Mult (Var "x", Nb 0))
```

Simplifier les expressions : à vous ....

Autre représentation possible

```
type binop = Add | Minus | Fois ;;

type expression = Var of string | Const of int
| Bin of expression*binop*expression;;

Bin (Var "x", Add, Bin (Const 2, Fois, Var "y"));;
```

*représente l'expression x + 2*y*

• Schémas de types récursifs particuliers

Modélisation d'un système arborescent de fichiers

Un fichier est :

- soit un fichier texte
- soit un répertoire contenant des fichiers

```
type fichier = Texte of string
             | Répertoire of (fichier list);;

let fichierex =
  Répertoire [
     Texte "let rec f x = x" ;
     Texte "%!PS-Adob ...";
     Répertoire [Texte "adjhklrjhzk" ; Texte ""] ;
     Texte "CV "
             ];;
```

```
let rec nb_texte f = match f with
  Texte _ -> 1
| Répertoire l -> nb_texte_list l
and
nb_texte_list l = match l with
  [] -> 0
| f::r -> (nb_texte f) + (nb_texte_list r);;
val nb_texte : fichier -> int = <fun>
val nb_texte_list : fichier list -> int = <fun>
```

⇒ 2 fonctions mutuellement récursives

```
nb_texte fichierex;;
- : int = 5
```

Types mutuellement récursifs

```
type homme = Adam | Fils of string*homme*femme | Inconnu
and femme = Eve | Fille of string*homme*femme | Inconnue;;

let abel = Fils ("Abel", Adam, Eve);;

let henoc = Fils("Henoc",Fils("Cain",Adam,Eve),Inconnue);;

let freres (h1, h2) = match (h1,h2) ->
  Fils(_, p1, m1), Fils(_,p2, m2) -> p1=p2 && m1=m2
| _ -> false;;
homme*homme -> bool
```