Programmation Fonctionnelle Avancée
8 : Variants polymorphes et sous-typage

Pierre Letouzey

Université Paris Cité
UFR Informatique
Institut de Recherche en Informatique Fondamentale
letouzey@irif.fr

3 avril 2023

Exemples (somme1.ml)

```
type num = Zero | S of num

let rec add n m = match n with
  | Zero -> m
  | S n -> S (add n m)

let two = S (S (Zero))

let _ = add two two
```

Exemples (somme2.ml)

```
(* the compiler warns about missing cases *)

let rec add n m = match n with
  | S n -> S (add n m)

let two = S (S (Zero))

let _ = add two two
```

Exemples (somme3.ml)

```
(* the compiler signals an error on unknown
     constructor *)

let rec add n m = match n with
  | Zero -> n
  | Succ n -> S (add n m)

let two = S (S (Zero))

let _ = add two two
```

Caractéristiques des types sommes

- Il faut déclarer le type avec ses constructeurs avant de les utiliser.
- Un constructeur appartient à *un seul* type somme ; si on déclare plusieurs types sommes ayant le même constructeur, seule la dernière déclaration est visible (même si OCaml fait un effort pour désambiguer les constructeurs dans certains cas).

Exemples (redef.ml)

```
type t1 = A | B
let x = B
type t2 = B | C
let y = B (* now of type t2 *)

let isB2 = function
  | B -> true
  | C -> false

let isB1 = function
  | A -> false
  | B -> true

let _ = isB1 x
let _ = isB2 y
```

Variants polymorphes, syntaxe

- Les *variants polymorphes* sont d'autres constructeurs qui peuvent être utilisés *sans avoir été déclarés préalablement* dans un type.
- Syntaxiquement, on les fait précéder d'une apostrophe à l'envers `` ` `` (backquote en V.O.)
- Un variant polymorphe tel que `` `A `` :
  - *appartient à différents types* : [`` `A|`B ] et [`` `A|`C|`G|`T] et ...
  - admet *différents types d'arguments* tant que le contexte de typage diffère : `` `A `` seul ou (`` `A 3 ``) ou (`` `A **true** ``) ou ...
- Au lieu d'un type exact de variants tel que [`` `A|`B], on va souvent rencontrer des *ensembles de types* convenables, exprimés via des bornes supérieures ($<$) et/ou inférieures ($>$).
- Cela induit une relation de sous-typage qui a donné le nom (variants polymorphes).

Exemples (variant1.ml)

```
let x = `A

let y = `A true

let z = `B 3

let xz = (`A, `B 3)

let l = [`A ; `B 3]
```

Variants polymorphes : borne inférieure

- Le « $>$ » signifie : « un type qui *contient au moins* les constructeurs suivants » ; c'est une *borne inférieure* qui apparaît quand on *produit une valeur* dont on sait qu'elle peut contenir au moins les constructeurs en question.
- La barre verticale signifie dans ce contexte « et » : [$>$ `A | `B] indique "contient au moins `A et `B".
- Un type ne peut pas contenir le même constructeur avec des types d'arguments différents dans un même contexte de typage.

Exemples (variant2.ml)

```
let ko = function
  | `A   -> 1
  | `A 3 -> 2

let ko = [`A; `A 3]

let ok = (`A, `A 5)
```

Interprétation des bornes inférieures

- Une borne inférieure est une *formule* qui décrit un ensemble de types possibles, via un tronc common de constructeurs qu'ils doivent tous avoir.
- On parle aussi parfois d'un *type ouvert* (moins exacte).
- [> `A | `B] est à lire comme : 'a tel que `A ∈ 'a ∧ `B ∈ 'a
- Donc [> `A | `B] inclut par exemple [`A|`B] et [`A|`B|`C] et [`A|`B|`Z|`Autre].
- Le type [> `A | `B] est *une instance* de [> `A ] ou [> `B] (car on a inclusion des ensembles de types correspondants).
- C'est une des raisons pour laquelle l'inférence de type utilise la *résolution de contraintes* et du *sous-typage* (à venir).

Exemples (variant3.ml)

```
(* borne superieure *)
let f = function
  | `B x -> x
  | `A x -> x

(* quelle est le type de cette fonction ? *)
let switch = function
  | `A -> `B
  | `B -> `A
```

Bornes supérieures

- Le « $<$ » signifie : « un type qui *peut contenir au plus* les constructeurs suivants » ; c'est une *borne supérieure* qui apparaît quand on *consomme une valeur* dont *on sait traiter seulement* les constructeurs en question.
- $[< \texttt{`A} \mid \texttt{`B}]$ est à lire comme :

  $$\texttt{'a tel que tout } \forall\, x \in \texttt{'a}, x = \texttt{`A} \vee x = \texttt{`B}$$

- Donc $[< \texttt{`A} \mid \texttt{`B}]$ inclut exactement $[\texttt{`A}]$ et $[\texttt{`B}]$ et $[\texttt{`A}|\texttt{`B}]$.

Exemples (variant4.ml)

```
let f = function `S a -> 1 | `Z -> 2

let g = function `S _ -> 4 | _ -> 5

let h = function `S _ -> 4 | x -> f x
```

Les bornes supérieures et inférieures simultanées

On peut se retrouver avec des bornes supérieures et inférieures à la fois :

- f a un $<$ parce qu'il n'accepte que ces deux cas.
  Un filtrage par motif "fermé" donne une borne supérieure.
- g a un $>$ grâce au cas _−> 5 : il accepte tout type qui peut contenir *au moins* `S.
  Un filtrage par motif "ouvert" donne une borne inférieure.
- [< `S **of** 'a | `Z > `S ] est une *conjonction* d'une borne inférieure [> `S] et d'une borne supérieure [< `S **of** 'a | `Z]
- h a un $>$ comme g, mais le x doit être acceptable pour f, d'où la même borne supérieure $<$ que pour f.

On peut nommer les types

- On peut utiliser les types de variants dans une définition de type.
- On ne peut pas utiliser les bornes directement dans les définitions : un type avec borne, aussi appelé *type ouvert* représente *un ensemble* de types (tous les types compris entre les bornes), et pas un seul type ; on ne peut donc pas les nommer.
- Par contre, on peut utiliser ces types ouverts quand on donne explicitement le type d'un identificateur.

Exemples (variant5.ml)

```
type good = [`A | `B of int]
type bad = [> `A | `B of int]

let f = function true -> `A | false -> `B
let fc = (f : bool -> [> `A | `B | `C])

let g = function `A -> 1 | `B -> 2
let gc = (g : [< `A] -> int)

let fcc = (f : bool -> [< `A | `C])
```

Exemples (variant6.ml)

```
(* lower bounds are combined by union *)
let (x: [> `A | `B ]) = `A
let y = (x: [> `A | `C])

(* upper bounds are combined by intersection *)
let (x: [< `A|`B|`C]) = `A
let y = (x: [< `A|`B|`D])

(* redundant constraints are dropped *)
let f = function `A -> 0 | `B -> 1
let g = (f: [<`A | `B | `C | `D] -> int)
```

Pour en savoir plus

Jacques Garrigue,
Polymorphic Variants.
Chapter 4.2 of the OCaml User's Manual.

Sous-typage

- Les types variants respectent une notion de *sous-typage* : un type variant fermé v est un *"sous-type"* d'un type variant fermé w si les constructeurs de v sont *inclus* dans les constructeurs de w.
- Autrement dit, v est un sous-type de w si toute valeur de type v est également du type w.
- C'est grâce aux variants polymorphes qu'une valeur peut avoir plusieurs types (à part du cas de la liste vide qui est traité de façon différente).

Exemples (sous1.ml)

```
type 'a vlist = [`Nil | `Cons of 'a * 'a vlist]

type 'a wlist = [`Nil | `Cons of 'a * 'a wlist
                | `Snoc of 'a * 'a wlist]

(* 'a vlist est un sous-type de 'a wlist *)

let x = `Cons (42,`Nil)
let y = (`Cons (42,`Nil) : int vlist)
let z = (`Cons (42,`Nil) : int wlist)
```

Contraintes de types et sous-typage

- Nous avons vu qu'on obtient une contrainte c (borne inférieure et/ou supérieure) pour le type de l'argument d'une fonction g (éventuellement même un type fermé).
- Si un type t_1 satisfait la contrainte c alors on peut appliquer la fonction g à toutes les valeurs de type t_1.
- Si t_2 est un sous-type de t_1 alors t_2 satisfait également la contrainte c, on peut donc aussi appliquer g à toutes les valeurs de type t_2.

Exemples (sous2.ml)

```
let rec g = function
  | `Nil -> true
  | `Cons (h,r) -> h=1 && (g r)
  | `Snoc (h,r) -> h=2 && (g r)

(* wlist satisfait les contraintes obtenues pour g
let z = (`Cons (42,`Nil) : int wlist)
let _ = g z

(* 'a vlist est un sous-type de 'a wlist *)
let y = (`Cons (42,`Nil) : int vlist)
let _ = g y
```

Cas d'usage : constructeurs surchargés

On veut plusieurs variantes d'une définition de type, et partager les constructeurs (les types sommes ne le permettent pas).
Exemple : dans la librairie Yojson, Yojson.Raw, Yojson.Safe et Yojson.Basic fournissent différents sous-ensemble des constructeurs suivants :

```
type json =
 [ `Assoc of (string * json) list          (*   S R *)
 | `Bool of bool                           (* B S R *)
 | `Float of float                         (* B S   *)
 | `Floatlit of string                     (*     R *)
 | `Int of int                             (* B S   *)
 | `Intlit of string                       (*   S R *)
 | `List of json list                      (* B S   *)
 | `Null                                   (* B S R *)
 | `String of string                       (* B S   *)
 | `Stringlit of string                    (*     R *)
 | `Tuple of json list                     (*   S R *)
 | `Variant of string * json option ]      (*   S R *)
```

Exemples (sous3.ml)

```
type t = [`A | `B]

let x = `A
let y = `C

(* types compatibles *)
let _ = (x = y)

(* types incompatibles *)
let _ = ((x:t) = y)
```

Coercion et Sous-typage

- On peut vouloir considérer une valeur de type `'a vlist` comme une valeur de type `'a wlist`.
- Dans ce cas on oublie que la `'a vlist` ne contient pas de `` `Snoc ``, mais on peut utiliser notre `'a vlist` *partout* où `'a wlist` est accepté.
- En OCaml, la coercion d'une valeur v d'un sous-type t1 du type t2 vers t2 se note `(v : t1 :> t2)`. Ici les parenthèses sont nécessaires, et on peut parfois omettre le `: t1`.

Exemples (sous4.ml)

```
type 'a vlist = [`Nil | `Cons of 'a * 'a vlist]
type 'a wlist = [`Nil | `Cons of 'a * 'a wlist
                      | `Snoc of 'a * 'a wlist]

let wlist_of_vlist  l = (l : 'a vlist :> 'a wlist)

let a : int vlist = `Cons (1, `Nil)

let _ = wlist_of_vlist a
```

Exemples (sous5.ml)

```
let open_vlist l = (l : 'a vlist :> [> 'a vlist])
let vl = (`Cons(73,`Nil) : int vlist)
let _ = open_vlist vl

let switch (x: [`A | `B]) =
  (match x with
   | `A -> `B
   | `B -> `A
   :> [`A | `B | `C])
```

Exemples (sous51.ml)

```
type t1 = [`A|`B]
type t2 = [`A|`B|`C]

let f = function
  | `A -> 0
  | `B -> 1
  | `C -> 2
  | _ -> 42

let (x:t1) = `A
let bad = f x
let ok = f (x :> t2)
```

Conversion, Contrainte, Coercion

Il ne faut pas confondre ces trois concepts :

- La *conversion* transforme une valeur d'un type en une valeur d'un autre type (par ex., un `int` en un `float`). Se fait en OCaml toujours explicitement, à l'aide des fonctions de conversion.
- Une *contrainte de type* restreint un type, en ajoutant des équations pour des variables de type. Elle peut être utile pour des expressions qui ont un type polymorphe.
- La *coercion* permet de considérer une valeur d'un type comme une valeur d'un type plus large. Elle peut être utile dans le contexte du sous-typage.

Exemples (coercion.ml)

```
type t1 = [`A | `B]
type t2 = [`A | `B | `C | `D ]

(* contraintes incompatibles *)
let x = (`A:t1)
let y = (x:t2)

(* une coercion ``ouvre'' le type *)
let y = (x :> t2)

(* contrainte redondante *)
let z = (x : [> `A])
```

Covariance et contravariance : produits, records, lists

- Comme les variants polymorphes introduisent du sous-typage, il faut savoir comment ce sous-typage se propage dans le reste des programmes.
- Le cas des produits :

$$\frac{\tau_1 < \tau_1' \qquad \tau_2 < \tau_2'}{\tau_1 \times \tau_2 < \tau_1' \times \tau_2'}$$

- Le cas des enregistrements : pareil
- Le cas des listes :

$$\frac{\tau_1 < \tau_1'}{\tau_1 \texttt{ list} < \tau_1' \texttt{ list}}$$

Covariance et contravariance : fonctions

- Le cas des fonctions :

$$\frac{\tau_1' < \tau_1 \qquad \tau_2 < \tau_2'}{\tau_1 \to \tau_2 < \tau_1' \to \tau_2'}$$

- Le type du résultat de la fonction suit la même direction de sous-typage que le type fonctionnel (on dit qu'il est *co-variant*, aussi noté avec un +), alors que le type du paramètre va dans le sens inverse (on dit qu'il est *contra-variant*, aussi noté avec un −).

D'où vient la contravariance ?

- Étant donnée une fonction $f : \tau_1 \to \tau_2$ quelconque.
- Quand peut-on aussi la considérer comme une fonction $\tau_1' \to \tau_2'$?
  1. Il faut que le résultat de f soit aussi du type τ_2' :

$$\tau_2 < \tau_2'$$

  2. Il faut que toute valeur de type τ_1' soit acceptée comme argument à la fonction f :

$$\tau_1' < \tau_1$$

Exemples (sous6.ml)

```
let f : [`T | `Z] -> int =
  function `T -> 1 | `Z -> 2

let g : [ `T] -> int =
  function `T -> 4

(* le type de f est un sous-type du type de g
   (en "oubliant" que f sait traiter `Z) *)
let ok = (f :> [ `T] -> int)

(* la reciproque n'est pas vraie
   (g ne sait pas gerer `Z) *)
let ko = (g :> [ `T | `Z] -> int)
```

Les annotations de variance sur les types abstraits

- Il est important de connaître la variance des paramètres des fonctions, pour savoir déterminer si un type fonctionnel est sous-type d'un autre.
- Le compilateur détermine cette information pour tous les types qu'il connaît, mais il ne peut pas le faire pour les types abstraits dans les interfaces des modules.

```
type 'a t
```

- C'est le programmeur qui peut indiquer la variance avec un + ou un -

```
type +'a t1
type -'a t2
```

Exemples (abstrait1.ml)

```
module type POS = sig
    type +'a t
end

(* OK *)
module P:POS = struct
  type 'a t = 'a*'a
end

(* Erreur *)
module N:POS = struct
  type 'a t = 'a->bool
end
```

Exemples (abstrait2.ml)

```
module type NEG = sig
  type -'a t
end

(* Erreur *)
module P:NEG = struct
  type 'a t = 'a*'a
end

(* OK *)
module N:NEG = struct
  type 'a t = 'a->bool
end
```

Exemples (sous8.ml)

```
let f1 = fun (`Nombre x) -> x = 0

let f2 = fun (`Nombre x) -> x = 0.0

(* Attention : type non habite ! *)
let f x = f1 x || f2 x
```

Types non habités

- Dans l'exemple du transparent précédent : La définition de f force le type de l'argument du constructeur `Nombre à être un int et un float à la fois (int & float).
- Dans cet exemple, cette contrainte est impossible à satisfaire. f est donc bien typée, mais on ne pourra jamais l'appliquer car il n'y a aucune valeur de ce type.

Résumé des variants polymorphes

- Les variants polymorphes permettent d'écrire des programmes plus flexibles que les types sommes habituels, et apportent de l'extensibilité.
- Mais ils ont leurs inconvénients :
  - ils induisent des problèmes de typage parfois *complexes*
  - le typage est moins strict, il devient possible d'utiliser un constructeur qui n'existe pas, ou parfois avec un mauvais type
  - ils induisent une petite perte d'efficacité des programmes.
- Donc : à utiliser seulement quand on peut en tirer profit.

Pour en savoir plus

Jacques Garrigue.
Code reuse through polymorphic variants.
In Workshop on Foundations of Software Engineering, 2000.
Available from `http://www.math.nagoya-u.ac.jp/~garrigue/papers/fose2000.html`.