

# Programmation Fonctionnelle Avancée

## 7 : Inférence de types, polymorphisme et traits impératifs

Pierre Letouzey

Université Paris Cité  
UFR Informatique  
Institut de Recherche en Informatique Fondamentale  
[letouzey@irif.fr](mailto:letouzey@irif.fr)

3 avril 2023

## Rappel sur le typage en OCaml

Les deux traits essentiels du système de typage de OCaml sont :

- ▶ Le *polymorphisme* : `List.map` manipule des listes de tout type. Les listes sont polymorphes, mais homogènes : dans une liste donnée, tous les éléments ont le même type.
- ▶ L'*inférence de types* : le système découvre tout seul le type le plus général, sans besoin de déclarer les types des identificateurs.

## Exemples (inf1.ml)

```
let _ = List.map
```

```
let _ = List.map (fun x -> x + 1) [1;2;3]
```

```
let _ = List.map (fun s -> s ^ "i") ["a"; "b"; "c"]
```

## Exemples (inf2.ml)

```
let k x y = x
```

```
let s x y z = (x y) (y z)
```

```
let f x y z = x (y z) (y , z)
```

## Inférence de types

- ▶ Comment est-ce que OCaml fait pour trouver le type le plus général d'un identificateur ?
- ▶ Regardons d'abord le dernier exemple du transparent précédant.
- ▶ Il s'agit d'un cas simple : pas de récurrence.
- ▶ On introduit une variable pour le type de chaque identificateur nouveau (ici : pour les identificateurs  $f, x, y, z$ ), et une variable pour chacune des expressions du côté droite :

$$\text{let } f \ x \ y \ z = x \ \overbrace{\underbrace{(y \ z)}_{e_2} \ \underbrace{(y, z)}_{e_3}}^{e_1}$$

- ▶ Variables :  $t_f, t_x, t_y, t_z, t_1, t_2, t_3$

## Système d'équations entre types

- ▶ Rappel,  $\rightarrow$  associe à droite :  $x \rightarrow y \rightarrow z = x \rightarrow (y \rightarrow z)$
- ▶ Pour **let**  $f\ x_1 \dots x_n = c$  :  
i.e. **let**  $f = \mathbf{fun}\ x_1 \dots x_n \rightarrow c$  :
  - ▶  $t_f = t_{x_1} \rightarrow \dots \rightarrow t_{x_n} \rightarrow t_c$
- ▶ Pour tous les sous-expressions  $e$  de  $c$ ,  $c$  incluse :
  - ▶ si  $e = (e_1, e_2) : t_e = t_{e_1} \times t_{e_2}$
  - ▶ si  $e = e_1\ e_2 \dots e_n : t_e = t_{e_1} \rightarrow \dots t_{e_n} \rightarrow t_e$

## Systèmes d'équations entre types

- On a :

$$\text{let } f \times y \ z = \overbrace{x \underbrace{(y \ z)}_{e_2} \underbrace{(y, z)}_{e_3}}^{e_1}$$

- Ceci donne les équations suivantes :

$$t_f = t_x \rightarrow t_y \rightarrow t_z \rightarrow t_1$$

$$t_x = t_2 \rightarrow t_3 \rightarrow t_1$$

$$t_y = t_z \rightarrow t_2$$

$$t_3 = t_y \times t_z$$

- Comment obtenir  $t_f$  à partir de ces équations ?

## Le sens des équations entre types

- ▶ Dans les équations il y a des variables de types, des constructeurs de types  $\rightarrow$  et  $\times$ , et éventuellement des constantes (`int`, `bool`).
- ▶ Propriétés des constantes et constructeurs de types :
  - ▶ Deux termes avec des constructeurs/constantes différentes en tête ne peuvent jamais être égaux.
  - ▶  $x_1 \rightarrow x_2 = y_1 \rightarrow y_2$  exactement si  $x_1 = y_1$  et  $x_2 = y_2$ . Pareil pour  $\times$ .
- ▶ Ce sont précisément les lois des symboles de fonctions non interprétées comme on les connaît en Logique !



## Resolution d'équations entre types

- ▶ L'algorithme pour résoudre des équations entre termes dans une structure de symboles de fonctions non interprétées est précisément l'algorithme d'*unification* de Herbrand/Robinson (voir un cours de *Logique*) !
- ▶ L'unification nous donne soit l'information que le système d'équations n'a pas de solution, soit une solution la plus générale (mgu) : toute solution peut être obtenue comme instance du mgu.
- ▶ Nous cherchons le type le plus général de  $f$  qui est permis par la définition de  $f$ , cela correspond exactement au mgu du système des équations.

## Rappel : L'algorithme d'unification (1)

- ▶ Donné :
  - ▶ Une signature  $\Sigma$  (un ensemble de symboles d'opérateurs avec leur arité). Dans le cas des équations de types on a

$$\Sigma = \{\rightarrow, \times, \text{int}, \text{bool}, \dots\}$$

où  $\rightarrow, \times$  sont d'arité 2 et  $\text{int}, \text{bool}$  sont d'arité 0.

- ▶ Un ensemble  $V$  de variables.
- ▶ Soit  $T(\Sigma, V)$  l'ensemble des termes construits sur  $\Sigma$  et  $V$
- ▶ Soit  $\text{free}(t)$  l'ensemble des variables présents dans le terme  $t$

## Rappel : L'algorithme d'unification (2)

- ▶ Voici les règles de transformation sur un système d'équations :
- ▶ *Decomposition* :

$$\frac{g(s_1, \dots, s_n) = g(t_1, \dots, t_n)}{s_1 = t_1, \dots, s_n = t_n}$$

Donc en pratique ici :

$$\frac{s_1 \rightarrow s_2 = t_1 \rightarrow t_2}{s_1 = t_1, s_2 = t_2} \qquad \frac{s_1 \times s_2 = t_1 \times t_2}{s_1 = t_1, s_2 = t_2}$$

- ▶ *Clash* :

$$\frac{g(s_1, \dots, s_n) = h(t_1, \dots, t_m)}{\text{false}}$$

quand  $g$  différent de  $h$  (p.ex.  $\times$  et  $\rightarrow$ )

## Rappel : L'algorithme d'unification (3)

- *Occur Check* :

$$\frac{x = t}{false}$$

quand  $x \in \text{free}(t)$ , et  $t$  est différent de  $x$ .

- *Variable Elimination* :

$$\frac{x = t \wedge \phi}{x = t \wedge \phi[x/t]}$$

quand  $x \notin \text{free}(t)$ ,  $x \in \text{free}(\phi)$ .

## Rappel : L'algorithme d'unification (4)

► *Variable Orientation*

$$\frac{t = x}{x = t}$$

quand  $t$  n'est pas une variable

► *Trivial Equation*

$$\frac{x = x}{true}$$

## Rappel : L'algorithme d'unification (5)

- ▶ Cet algorithme termine toujours : soit il donne *false*, soit un système d'équations en forme normale (aucune transformation ne s'applique).
- ▶ Le résultat est *équivalent* au système d'origine.
- ▶ Quand l'algorithme se termine avec un résultat différent de *false* : on a un système d'équations de la forme

$$\begin{array}{ccc} x_1 & = & t_1 \\ & \vdots & \\ x_n & = & t_n \end{array}$$

où aucun des  $x_i$  n'apparaît dans les termes  $t_j$ .

- ▶ Variantes : séparation entre équations résolues et non résolution, calcul d'une solution sous forme triangulaire.

## Exemple : résolution d'équations (1)

► Système de départ :

$$t_f = t_x \rightarrow (t_y \rightarrow (t_z \rightarrow t_1))$$

$$t_x = t_2 \rightarrow (t_3 \rightarrow t_1)$$

$$t_y = t_z \rightarrow t_2$$

$$t_3 = t_y \times t_z$$

► Élimination de  $t_3$  :

$$t_f = t_x \rightarrow (t_y \rightarrow (t_z \rightarrow t_1))$$

$$t_x = t_2 \rightarrow ((t_y \times t_z) \rightarrow t_1)$$

$$t_y = t_z \rightarrow t_2$$

$$t_3 = t_y \times t_z$$

## Exemple : résolution d'équations (2)

- Élimination de  $t_y$  :

$$t_f = t_x \rightarrow ((t_z \rightarrow t_2) \rightarrow (t_z \rightarrow t_1))$$

$$t_x = t_2 \rightarrow (((t_z \rightarrow t_2) \times t_z) \rightarrow t_1)$$

$$t_y = t_z \rightarrow t_2$$

$$t_3 = (t_z \rightarrow t_2) \times t_z$$

- Élimination de  $t_x$  :

$$t_f = (t_2 \rightarrow (((t_z \rightarrow t_2) \times t_z) \rightarrow t_1) \rightarrow ((t_z \rightarrow t_2) \rightarrow (t_z \rightarrow t_1)))$$

$$t_x = t_2 \rightarrow (((t_z \rightarrow t_2) \times t_z) \rightarrow t_1)$$

$$t_y = t_z \rightarrow t_2$$

$$t_3 = (t_z \rightarrow t_2) \times t_z$$



## Exemple : résolution d'équations (4)

- Le mgu associe donc à  $t_f$  le type suivant :

$$(t_2 \rightarrow (((t_z \rightarrow t_2) \times t_z) \rightarrow t_1)) \rightarrow ((t_z \rightarrow t_2) \rightarrow (t_z \rightarrow t_1))$$

- Enlevons les parenthèses inutiles et renommons les variables comme OCaml ( $t_2 \mapsto a$ ,  $t_z \mapsto b$ ,  $t_1 \mapsto c$ ) :

$$(a \rightarrow (b \rightarrow a) \times b \rightarrow c) \rightarrow (b \rightarrow a) \rightarrow b \rightarrow c$$

## Exemple d'échec de l'inférence de type

- Regardons un deuxième exemple :

```
let f g = (g 42) && (g "truc")
```

- Système d'équations :

$$t_f = t_g \rightarrow \text{bool}$$

$$t_g = \text{int} \rightarrow \text{bool}$$

$$t_g = \text{string} \rightarrow \text{bool}$$

- On obtient avec les règles d'unification :

$$\text{int} \rightarrow \text{bool} = \text{string} \rightarrow \text{bool}$$

$$\text{int} = \text{string} \quad \textcolor{red}{\downarrow}$$

## Pourquoi cet echec ?

- ▶ `let f g = (g 42) && (g "truc")`
- ▶ Pourquoi est-ce que  $g$  ne peut pas être du type  $: 'a \rightarrow \text{bool}$  ?
- ▶ Pour voir la réponse il faut expliciter les quantificateur des variables de types.

## Quantification de variables de types

- ▶ Toutes les variables dans un type sont quantifiées universellement au début du type :
- ▶ Par exemple : Un type comme

$$a \rightarrow b \rightarrow a \times (b \rightarrow a)$$

est à lire comme

$$\forall a, b : \quad a \rightarrow b \rightarrow a \times (b \rightarrow a)$$

- ▶ Les variables  $a, b$  peuvent être instanciées par des types quelconques.

## Quantification de variables de types

- Reprenons l'exemple :

```
let f g = (g 42) && (g "truc")
```

- Le type qu'on essaye de construire ici est :

$$(\forall a : a \rightarrow \text{bool}) \rightarrow \text{bool}$$

- Des types avec des  $\forall$  sous une flèche n'existent pas en OCaml, l'inférence a raison de refuser cette définition. Ce qui existe en OCaml est le type

$$\forall a : ((a \rightarrow \text{bool}) \rightarrow \text{bool})$$

mais c'est un type différent !

## Quantification de variables de types

- ▶ *Normalement*, dans tous les types les variables de types sont (implicitement) quantifiées  $\forall$ , avec un quantificateur devant le type complet.
- ▶ *Normalement*, les variables de types libres (non quantifiées) paraissent seulement pendant la résolution des équations, une fois le type le plus général obtenu les variables sont quantifiées.
- ▶ Dans **let**  $f \ x_1 \ \dots \ x_n = e$ , toutes les nouvelles variables du type de  $f$  sont quantifiées au début par  $\forall$ .
- ▶ Nous verrons une exception à cette règle un peu plus tard.

## Example : Quantification de variables de type (1)

$$\text{let } f \ g \ h = \text{fun } x \rightarrow \underbrace{h \ (\underbrace{g \ x}_{t_3})}_{t_1}^{t_2}$$

► On obtient le système d'équations :

$$t_f = t_g \rightarrow (t_h \rightarrow t_1)$$

$$t_1 = t_x \rightarrow t_2$$

$$t_h = t_3 \rightarrow t_2$$

$$t_g = t_x \rightarrow t_3$$

## Exemple : Quantification de variables de type (2)

- La solution trouvée pour la variable  $t_f$  est :

$$t_f = (t_x \rightarrow t_3) \rightarrow (t_3 \rightarrow t_2) \rightarrow (t_x \rightarrow t_2)$$

- Dans ce type, les variables  $t_x$ ,  $t_2$ ,  $t_3$  sont libres, elles sont donc implicitement quantifiées avec un  $\forall$
- Le type obtenu pour  $f$  est donc à lire comme :

$$\forall t_x, t_3, t_2 : (t_x \rightarrow t_3) \rightarrow (t_3 \rightarrow t_2) \rightarrow (t_x \rightarrow t_2)$$



## Et la récursivité ?

- ▶ Comment typer **let rec**  $f\ x_1\ \dots\ x_n = e$  ?
- ▶ On procède comme avant, sauf que  $e$  peut contenir des appels récursifs à  $f$ .
- ▶ Donc  $t_f$  peut apparaître dans les équations de types concernant les expressions  $e_i$ .
- ▶ On résout ensuite par unification comme auparavant.
- ▶ Exemple : **let rec**  $f\ n = \text{if } n = 0 \text{ then } 0 \text{ else } n + f\ (n-1)$ 
  - ▶ Quelles sont les équations ?
  - ▶ Comment s'unifient-elles pour obtenir  $t_f = \text{int} \rightarrow \text{int}$  ?

## Quelques résultats fondamentaux

- ▶ Il existe un algorithme qui, étant donnée une expression  $e$ , trouve, si elle est typable, son type  $\sigma$  *le plus général possible*, aussi appelé *type principal*.
- ▶ Le premier algorithme pour cela est le W de *Damas et Milner*, qu'on trouve dans *Principal type-schemes for functional programs. 9th Symposium on Principles of programming languages (POPL'82)*.
- ▶ Cet algorithme utilise de façon essentielle l'algorithme d'unification de Herbrand/Robinson.
- ▶ Les algorithmes modernes utilisent plutôt directement la résolution de contraintes.
- ▶ À la surprise générale, en 1990 on a montré que l'inférence de type pour le noyau de ML est DEXPTIME complète.

## Exemples (inf4.ml)

```
let p x y = fun z -> z x y
```

```
let test x0 =  
  let x1 = p x0 x0 in  
  let x2 = p x1 x1 in  
  let x3 = p x2 x2 in  
  let x4 = p x3 x3 in  
  let x5 = p x4 x4 in  
  let x6 = p x5 x5 in  
  let x7 = p x6 x6 in  
  x7
```

## Comment faire exploser le typeur d'OCaml...

- ▶ Dans l'exemple précédent, le type de `xk` est de taille  $2^k$  !
- ▶ On peut même avoir une double exponentielle :

```
let boom z =
  let f0 = fun x -> (x, x) in
  let f1 = fun y -> f0 (f0 y) in
  let f2 = fun y -> f1 (f1 y) in
  let f3 = fun y -> f2 (f2 y) in
  let f4 = fun y -> f3 (f3 y) in
  f4 z
```

- ▶ Heureusement, en pratique, personne n'écrit de code ainsi.
- ▶ L'inférence de type à la ML reste un des systèmes de type les plus puissants.

## Les règles de typage d'OCaml

- ▶ Standard :
  - ▶ sommes
  - ▶ tuples
  - ▶ enregistrements
- ▶ Plus compliqué : *value restriction*
  - ▶ typage des effets de bord (ce chapitre)
- ▶ Avancé :
  - ▶ modules
  - ▶ récursion polymorphe
  - ▶ objets
  - ▶ *variants polymorphes* (voir la semaine prochaine)
  - ▶ *GADT* (voir dans deux semaines)

## La value restriction

- ▶ En OCaml, on dispose de structures mutables capables de contenir des données de tout type.
- ▶ Opérateurs pour les références :
 

$\text{ref} \quad \forall a : a \rightarrow (a \text{ ref})$	créer une référence vers une valeur
$! \quad \forall a : (a \text{ ref}) \rightarrow a$	déréférencer
$:= \quad \forall a : (a \text{ ref}) \rightarrow a \rightarrow \text{unit}$	changer la valeur d'une case mémoire référencée
- ▶ Ici `ref` est un constructeur de type (au même titre que  $\rightarrow$ ,  $\times$ , `list`, etc.)
- ▶ Essayons d'appliquer notre algorithme d'inférence de types en présence de références.

## Exemples (inf6.ml)

```
let c = ref (fun x -> x)
```

```
let _ = c := (fun x -> x+1)
```

```
let _ = !c true
```

## Inférence de types en présence de références

- ▶ Selon notre algorithme, `ref (fun x -> x)` a le type

$$(a \rightarrow a) \text{ ref}$$

- ▶ On obtient donc pour `c` le type :

$$\forall a : (a \rightarrow a) \text{ ref}$$

- ▶ Le quantificateur universel pour `a` va permettre d'instancier `a` une fois par `int`, et puis par `bool`.
- ▶ Évidemment OCaml a raison de refuser ce code, il y a donc un problème avec notre inférence de types.



## Restriction de quantification

- ▶ Le souci vient du fait que la variable 'a est quantifiée avec  $\forall$ .
- ▶ Ici, une fois la variable 'a instanciée, on ne devrait plus avoir le droit de changer cette instanciation.
- ▶ En présence de traits impératifs, on ne peut donc pas quantifier les variables dans les types comme avant.
- ▶ Idée : les variables de types sont quantifiées seulement quand l'expression à la droite du `let` satisfait certaines conditions, sinon la variable reste *libre* et peut donc être instanciée une seule fois.
- ▶ Ces conditions restent à déterminer !
- ▶ OCaml affiche une variable de type libre comme `'__weak` (ou bien `'__a` dans les versions plus anciennes)

## Inférence de type en présence de références

- ▶ Quelles sont les conditions qui permettent de quantifier les variables de type ?
- ▶ Une première idée est : l'expression ne contient pas du tout de références.
- ▶ Ça marche, mais a une conséquence assez grave : le polymorphisme est effectivement désactivé dès qu'on utilise des références dans une fonction, même si l'utilisation est parfaitement sûre.
- ▶ Regardons l'exemple suivant :

## Exemples (value-restriction3.ml)

```

let fastrev = function list ->
  let left = ref list
  and right = ref []
  in begin
    while !left <> [] do
      right := (List.hd (!left)) :: !right;
      left := List.tl (!left)
    done;
    !right
  end

```

(\* OK ! \*)

```

let _ = fastrev [1;2;3;4]

```

```

let _ = fastrev [true; true; false; false]

```

## Inférence de type en présence de références

- ▶ La question est alors : trouver la bonne condition sous laquelle les variables de type peuvent être quantifiées, tel que :
  - ▶ les erreurs de type pendant l'exécution du programme sont exclues ;
  - ▶ les fonctions polymorphes utilisant les références de façon sûre restent autorisées.
- ▶ Cette question a donné lieu à plusieurs propositions, toutes assez complexes.
- ▶ Une solution simple a été trouvée par Andrew K. Wright en 1995.

## La *Value Restriction*

Solution simple introduite par SML : permettre la généralisation *seulement* des valeurs (d'où le nom). Les valeurs sont :

- ▶ les constantes (13, "foo", 13.0, ...)
- ▶ les variables (x, y, ...)
- ▶ les fonctions (fun x -> e), où e une expression *quelconque*
- ▶ les constructeurs appliqués à des valeurs (Foo v)
- ▶ un n-uplet de valeurs (v1, v2, ...)
- ▶ un enregistrement contenant seulement des valeurs {l1 = v1, l2 = v2, ...}
- ▶ une liste de valeurs [v1, v2, ...]
- ▶ mais *pas* une application (f e), et en particulier pas (**ref** e).

## Exemples (value-restriction1.ml)

```
let c = ref (fun x -> x)
    (* type of c : ('_weak1 -> '_weak1) ref
       where here '_weak1 is a free variable! *)
```

```
let _ = c := (fun x -> x+1)
let _ = c
    (* type of c : (int -> int) ref, '_weak1 has been
       instantiated *)
```

```
let _ = !c true
    (* type error : clash between int and bool *)
```

## Conséquences de la Value Restriction

- ▶ Dans l'exemple précédent, `c` a le type `('__weak1 -> '__weak1) ref`
- ▶ Explication : l'expression n'est pas une valeur, donc il n'y a pas de généralisation ( $\forall$ ) lors du `let`.
- ▶ Inconvénient : il y a parfois des programmes correctes qui sont refusés, comme sur l'exemple suivant.
- ▶ On peut souvent contourner le problème.
- ▶ OCaml utilise une solution légèrement plus générale, due à Jacques Garrigue.

## Exemples (value-restriction2.ml)

```
let id = fun x -> x
```

```
(* type error *)
```

```
let f = id id
```

```
let _ = f 42
```

```
let _ = f "truc"
```

```
(* solution: eta-expansion  
  (i.e. expliciter l'argument) *)
```

```
let g = fun x -> (id id) x
```

```
let _ = g 42
```

```
let _ = g "truc"
```



## Limites de la méthode I

```

let twice_only f =
  (* yields a variant of f that can be applied twice *)
  (* only, and that behaves like identity after that. *)
  let counter = ref 0 in
  fun x →
    counter := !counter+1;
    if !counter <= 2 then f x else x
    
```

```

(* the function double is not polymorphic *)
(* since the right-hand side is not a value. *)
let double = twice_only (fun x → x@x)
    
```

```

let _ = double [1; 2]
let _ = double [1; 2]
let _ = double [1; 2]
let _ = double ["a"; "b"]
    
```

## Limites de la méthode II

```
(* Using eta-expansion we get a polymorphic *)
(* function , but it does not behave the same! *)
(* At each application of double_eta , a new *)
(* counter is created. *)
```

```
let double_eta =
  fun y -> twice_only (fun x -> x@x) y
```

```
let _ = double_eta [1; 2]
let _ = double_eta [1; 2]
let _ = double_eta [1; 2]
let _ = double_eta ["a"; "b"]
```

## Pour en savoir plus



Harry G. Mairson.

Deciding ML typability is complete for deterministic exponential time.

In Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '90, pages 382–401, New York, NY, USA, 1990. ACM.



Andrew K. Wright.

Simple imperative polymorphism.

Lisp Symb. Comput., 8(4) :343–355, December 1995.



Jacques Garrigue.

Relaxing the value restriction.

In FLOPS 2004, pages 196–213.