

Programmation Fonctionnelle Avancée 9 : Types fantômes et GADT

Pierre Letouzey

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

3 avril 2023

© Roberto Di Cosmo et Ralf Treinen et Pierre Letouzey

Exemples (phantom1.ml)

```
type 'a t = float

let (x: char t) = 3.0
let (y: string t) = 5.0

let _ = x+.y
```

Les types fantômes (phantom types)

- Un type fantôme est un type polymorphe dont au moins un paramètre de type n'est pas utilisé dans sa définition. En voici un :

```
type 'a t = float
```

- Tant que les définitions restent visibles, toutes les instances des types fantômes sont équivalentes pour le compilateur.
- Les égalités `char t = float = string t` sont connues du typeur, qui ne retient que le corps de la définition : `float`

Les types fantômes (phantom types)

- Les choses deviennent bien plus intéressantes si ces égalités se retrouvent *cachées* par la définition d'un type abstrait dans l'interface d'un module.
- Cela permet par exemple d'avoir deux “versions” différentes du type `float` pour des unités de mesure différentes.

Exemples (phantom2.ml)

```

module type LENGTH = sig
  type 'a t
  val meters : float -> ['Meters] t
  val feet : float -> ['Feet] t
  val (+.) : 'a t -> 'a t -> 'a t
  val to_float : 'a t -> float
end

module Length:LENGTH = struct
  type 'a t = float
  let meters f = f
  let feet f = f
  let (+.) = Stdlib.(+.)
  let to_float f = f
end
    
```

Même les meilleurs peuvent en avoir besoin



Exemples (phantom3.ml)

```

open Length

let m1 = meters 10.
let f1 = feet 40.

let _ = m1 +. m1
let _ = f1 +. f1
let _ = to_float (f1 +. f1)
let _ = m1 +. f1
    
```

Utiliser les types fantômes

- ▶ On peut essayer de pousser cet exemple plus loin, et coder des informations plus fines dans le paramètre fantôme.
- ▶ Dans l'exemple suivant, nous essayons d'appliquer ce principe pour raffiner un type de listes en "listes vides" et "listes non vides".
- ▶ Nous définissons deux types abstraites `vide` et `nonvide` qui nous servent seulement pour faire cette distinction.

```

module type LISTE = sig
  type vide
  type nonvide
  type ('vide_ou_non, 'e) t
  val listevide : (vide, 'e) t
  val cons: 'e → ('vide_ou_non, 'e) t → (nonvide, 'e) t
  val head: (nonvide, 'e) t → 'e
end

module Liste:LISTE = struct
  type vide
  type nonvide
  type ('vide_ou_non, 'e) t = 'e list
  let listevide = []
  let cons v l = v::l
  let head = function [] → assert false | a::_ → a
end
    
```

Listes vides et non vides

- ▶ L'exemple précédent permet de distinguer les listes vides et non vides au *niveau du typage*.
- ▶ Problème : quel est le type de la fonction `tail` ?
- ▶ Son argument doit être du type `(nonvide, 'e) t`. Mais le résultat peut être vide ou non vide.
- ▶ Si on utilise des variants polymorphes comme paramètres, le sous-typage des variants polymorphes donne des types plus précis pour les constructeurs.

Exemples (phantom5.ml)

open Liste

```

let _ = listevide
let _ = cons 3 listevide
let _ = head (cons 3 listevide)
    
```

```

(* erreur de typage ! *)
let _ = head listevide
    
```

```

module type LISTEVP = sig
  type ('vide_ou_non, 'e) t
  val listevide : ([ `Vide ], 'e) t
  val cons : 'e → ([< `Vide | `Nonvide ], 'e) t
           → ([ `Nonvide ], 'e) t
  val head : ([ `Nonvide ], 'e) t → 'e
  val tail : ([ `Nonvide ], 'e) t →
             ([< `Vide | `Nonvide ], 'e) t
end

module ListeVP:LISTEVP = struct
  type ('vide_ou_non, 'e) t = 'e list
  let listevide = []
  let cons v l = v::l
  let head = function [] → assert false | a::_ → a
  let tail = function [] → assert false | _::t → t
end
    
```

Exemples (phantom7.ml)

```
open ListeVP

let _ = listevide
let x = cons 3 listevide
let _ = head x
let _ = head listevide (* type error *)

let _ = tail x

let _ = tail (tail x) (* assert failure *)
```

```
type zero          (* type comptant 0 *)
type 'nat succ     (* type comptant +1 *)

module type LISTECOUNT = sig
  type ('lon, 'ele) t
  val listevide : (zero, 'ele) t
  val estvide : ('lon, 'ele) t -> bool
  val cons : 'ele -> ('lon, 'ele) t -> ('lon succ, 'ele) t
  val head : ('lon succ, 'ele) t -> 'ele
  val tail : ('lon succ, 'ele) t -> ('lon, 'ele) t
end

module Listecount:LISTECOUNT = struct
  type ('a, 'b) t = 'b list
  let listevide = []
  let estvide = function [] -> true | _ -> false
  let cons v l = v::l
  let head = function [] -> assert false | a::_ -> a
  let tail = function [] -> assert false | _::l -> l
end
```

Pourquoi cette erreur ?

- ▶ Notre système de type est maintenant trop faible pour éviter des erreurs d'applications de fonctions à des listes vides.
- ▶ La distinction en *vide* et *non-vidé* est trop grossière.
- ▶ Idée : coder la longueur de la liste dans le type !
- ▶ Pour faire cela nous devons représenter chaque nombre naturel par un *type* différent : nous utilisons un type zero, et un constructeur de type 'nat succ.
- ▶ Ainsi, le nombre naturel 2 est représenté par le type zero succ succ (notation postfix pour les applications de constructeurs de type).

Exemples (phantom9.ml)

```
open Listecount

let _ = head listevide

let l4 = cons 1 (cons 2 (cons 3 (cons 4 listevide)))

let l1 = tail (tail (tail l4))

let _ = head l1

let _ = tail (tail l1)
```

Listes avec codage de longueur

- ▶ Les deux derniers transparents ont poussé l'exercice encore plus loin : on encode la longueur de la liste dans un type fantôme (en codage unaire).
- ▶ L'utilité est limitée : on peut encore écrire map avec ces types de données, mais pas la fonction append (pourquoi?).

Cas d'utilisation : contrôle d'accès dans libvirt

- ▶ Libvirt is a C toolkit to interact with the virtualization capabilities of recent versions of Linux (and other OSes). The library aims at providing a long term stable C API for different virtualization mechanisms. It currently supports QEMU, KVM, XEN, OpenVZ, LXC, and VirtualBox.
- ▶ Libvirt utilise des types fantômes pour le contrôle d'accès aux ressources virtualisées.
- ▶ Voir : <http://camltastic.blogspot.fr/2008/05/phantom-types.html>

Cas d'utilisation : Lablgtk2

On utilise les types fantômes pour typer les widgets (avec des variants polymorphes) :

```
type (→ 'a) gtkobj
type widget = [ `widget ]
type container = [ `widget | `container ]
type box = [ `widget | `container | `box ]
```

Cela autorise la coercion sûre entre les classes :

```
(mybox : box gtkobj :> container gtkobj)
```

Voir le [manuel](#) Lablgtk et ce [message](#) de Jacques Garrigue (caml-list, 14/9/2001).

Exemples (libvirt1.ml)

```
module type CONNECTION = sig
  type 'a t
  val connect_readonly : unit → [ `Readonly ] t
  val connect : unit → [ `Readonly | `Readwrite ] t
  val status : [ > `Readonly ] t → int
  val destroy : [ > `Readwrite ] t → unit
end

module Connection : CONNECTION = struct
  type 'a t = int
  let count = ref 0
  let connect_readonly () = incr count; !count
  let connect () = incr count; !count
  let status c = c
  let destroy c = ()
end
```

Exemples (libvirt2.ml)

```
open Connection
let conn_ro = connect_readonly ()
let _ = status conn_ro
let _ = destroy conn_ro (* error *)

let conn_rw = connect ()
let _ = status conn_rw
let _ = destroy conn_rw
```

Exemples (html2.ml)

```
open Html
let x = div [ pdata "" ]

(* erreur, pas de div dans les span *)
let x' = span [ div [] ]

(* OK *)
let x'' = span [ span [] ]
let f s = div [ s; pdata "" ]
let f' s = div [ s; span [] ]
let f'' s = div [ s; span []; pdata "" ]
```

```
module type HTML = sig
  type +'a elt
  val pdata : string -> [> `Pdata ] elt
  val span : [< `Span | `Pdata ] elt list
              -> [> `Span ] elt
  val div : [< `Div | `Span | `Pdata ] elt list
              -> [> `Div ] elt
end
module Html : HTML = struct
  type raw =
    | Node of string * raw list
    | Pdata of string
  type 'a elt = raw
  let pdata s = Pdata s
  let div l = Node ("div", l)
  let span l = Node ("span", l)
end
```

Bilan des types fantômes

avantages :

- ▶ *étiquettes* sur les types qui permettent
- ▶ un contrôle *statique* du bon usage des données
- ▶ sans aucune pénalité à l'exécution (les types sont effacés à la compilation)

limitations : expressivité limitée

Pour en savoir plus



Daan Leijen and Erik Meijer.

Domain specific embedded compilers.

[SIGPLAN Not.](#), 35(1) :109–122, December 1999.

Exemples (gadt1.ml)

```
let silly x = match x with
| [] -> 1
| a::r -> match x with a'::r' -> 2

(* pattern matching non exhaustive *)
```

Limitation des types algébriques

- ▶ Il y a des situations dans lesquelles *on sait* qu'un filtrage est complet, mais le compilateur OCaml ne le voit pas.
- ▶ C'est le cas dans le code (simplifié) suivant : dans la deuxième branche, nous savons que $x = a::r$, donc le deuxième match est exhaustive.

Les GADT

- ▶ De l'anglais : *Generalised Algebraic Data Types*
- ▶ Il s'agit d'une extension du langage (contrairement aux types fantômes qui sont une astuce de programmation).
- ▶ Début dans OCaml 4.00 (2012), stabilisés un peu après.
- ▶ Première étape de l'extension : dans la définition d'un type algébrique, chaque constructeur peut venir avec son type complet (arguments *et* type de retour).
- ▶ Pour cela, une syntaxe alternative (et plus générale) permet de déclarer le type complet d'un constructeur.

Syntaxe alternative des types algébriques

- La syntaxe vue jusqu'ici :

```
type tree =  
  | Leaf of int  
  | Node of tree * int * tree
```

- Nouvelle syntaxe, avec type résultat des constructeurs :

```
type tree =  
  | Leaf : int -> tree  
  | Node : tree * int * tree -> tree
```

- Ca devient plus intéressant quand le type est polymorphe!
- Deuxième étape de l'extension : le type résultat d'un constructeur peut être une *instance* du type qu'on est en train de définir (souvent via des paramètres plus précis).

Exemples (gadt3.ml)

```
(* On peut aider le typeur, et lui dire  
   que tail_or_empty est une fonction ayant  
   un type polymorphe. Chaque cas peut donc  
   avoir une instance différente de ce type.  
   *)
```

```
let head_or_one: type v . v ilist -> int =  
  function  
    | Nil -> 1  
    | (Cons _) as l -> head l
```

Exemples (gadt2.ml)

```
type empty  
type nonempty
```

```
type 'empty_or_not ilist =  
  | Cons : int * 'v ilist -> nonempty ilist  
  | Nil : empty ilist
```

```
(* jusqu'ici, tout va bien *)  
let head = function Cons(x,r) -> x
```

```
(* mais le typeur exige des motifs du meme type *)  
let head_or_one = function  
  | Nil -> 1  
  | (Cons _) as l -> head l
```

Exemples (gadt4.ml)

```
(* Version de silly qui ne donne plus de warning *)
```

```
let silly : type v . v ilist -> int = function  
  | Nil -> 1  
  | (Cons _) as l ->  
    (* here l has type nonempty ilist *)  
    match l with Cons (x,_) -> 2
```


Exemples (gadt5.ml)

(Exemple: les listes vides/non-vides, encore *)*

```
type empty  
type nonempty
```

```
type ('empty_or_not, 'e) t =  
  | Nil : (empty, 'e) t  
  | Cons : 'e * ('empty_or_not, 'e) t ->  
            (nonempty, 'e) t
```

```
let hd = function Cons (x, r) -> x
```

Exemples réalistes : interpréteur

- ▶ Interpréteur pour un petit langage d'expressions
- ▶ Commençons une première version sans GADT
- ▶ Types de base int, bool, plus des paires de types.
- ▶ On veut avoir un seul type d'expressions
- ▶ Il faut ici aussi un type résultat pour l'évaluation

Exemples (gadt6.ml)

```
let rec length : type v e. (v, e) t -> int  
= function  
  | Nil -> 0  
  | Cons (_, r) -> 1 + length r
```

```
let rec map :  
  type v e1 e2. (e1 -> e2) -> (v, e1) t -> (v, e2) t  
= fun f -> function  
  | Nil -> Nil  
  | (Cons (x, r)) -> Cons(f x, map f r)
```

Exemples (gadt8.ml)

(expressions sans GADT, de l'OCaml classique *)*

```
type expr =  
  | Int of int  
  | Bool of bool  
  | Pair of expr * expr  
  | IfThenElse of expr * expr * expr
```

```
(* valeurs *)  
type res = I of int | B of bool | P of res * res
```

Exemples (gadt9.ml)

```
let rec eval = function
| Int x -> I x
| Bool b -> B b
| Pair (e1,e2) -> P (eval e1, eval e2)
| IfThenElse (e1,e2,e3) ->
  match eval e1 with
  | B b -> if b then eval e2 else eval e3
  | _ -> failwith "Boolean expected in an if!"

(* exception pendant l'exécution : *)
let _ = eval (IfThenElse (Int 0, Int 1, Int 2))
(* OK : *)
let _ = eval (IfThenElse (Bool true, Int 3, Int 4))
let _ =
  eval (IfThenElse (Bool true, Int 2, Bool true))
```

Exemples (gadt10.ml)

```
(* expression avec GADT *)

type _ expr =
| Int : int -> int expr
| Bool : bool -> bool expr
| Pair : 'a expr * 'b expr -> ('a * 'b) expr
| IfThenElse : bool expr * 'a expr * 'a expr
               -> 'a expr
```

Limitations de l'interpréteur sans GADT

Comme on ne sait pas classer les expressions par leur type, on est obligé :

- ▶ d'aplatir les expressions dans un type `expr`
- ▶ d'aplatir les résultats dans un type `res`
- ▶ d'écrire la fonction `eval` en traitant des cas d'erreur à l'exécution

Exemples (gadt11.ml)

```
let rec eval : type a. a expr -> a = function
| Int x -> x
| Bool b -> b
| Pair (e1,e2) -> (eval e1, eval e2)
| IfThenElse (b,e1,e2) ->
  if eval b then eval e1 else eval e2

(* Erreurs de typage ! *)
let _ = eval (IfThenElse (Int 0, Int 1, Int 2))
let _ =
  eval (IfThenElse (Bool true, Int 1, Bool true))
(* Ok *)
let _ = eval (IfThenElse (Bool true, Int 1, Int 2))
```

Exemples réalistes : fonctions composables

- Comment typer une suite $[f_1; f_2; f_3; \dots; f_n]$ contenant des fonctions qui se composent ?

$$A_1 \xrightarrow{f_1} A_2 \xrightarrow{f_2} A_3 \xrightarrow{f_3} \dots \xrightarrow{f_n} A_{n+1}$$

- Le type du résultat d'une fonction dans une telle suite doit être le type de l'argument de la fonction suivante.
- Cela semble impossible avec le langage OCaml sans GADT.
- Avec des GADT, on peut donner un type précis qui permet d'écrire du code bien typé.

Variables de types et GADT

- Dans cet exemple, on voit qu'on peut utiliser des variables de type qui n'apparaissent pas dans le résultat du constructeur : c'est le cas du 'b dans le type de Seq
- Il s'agit de variables dites *existentielles*, parce que le type de Seq :

$$\forall a, b. (a \rightarrow b) * (b, c) \text{ seq} \rightarrow (a, c) \text{ seq}$$

peut se lire comme :

$$\forall a, c. \{ \exists b. (a \rightarrow b) * (b, c) \text{ seq} \} \rightarrow (a, c) \text{ seq}$$

Exemples (gadt12.ml)

```
type (_,_) seq =
  | Id : ('a, 'a) seq
  | Seq : ('a -> 'b) * ('b, 'c) seq -> ('a, 'c) seq

let rec apply : type a b. (a, b) seq -> a -> b =
  function
  | Id -> fun x -> x (* here a = b *)
  | Seq (f, s) -> fun x -> x |> f |> apply s

let bad = Seq(truncate, Seq(string_of_float, Id))
let good = Seq(truncate, Seq(string_of_int, Id))
let _ = apply good 3.5
```

Exemples réalistes : arbres binaires complets

Un type d'arbres binaires complets : toutes les racines ont la même profondeur 'h.

```
type zero = unit (* codage des entiers en type, *)
type 'n succ = 'n * unit (* concret cette fois-ci *)

type ('a, 'h) tree =
  | Leaf : ('a, zero) tree
  | Node : ('a, 'h) tree * 'a * ('a, 'h) tree -> ('a, 'h succ) tree

let good = Node(Node(Leaf, 1, Leaf), 3, (Node(Leaf, 2, Leaf)))
let bad = Node(Leaf, 1, Node(Leaf, 2, Leaf))

let rec map : type a b h. (a -> b) -> (a, h) tree -> (b, h) tree
= fun f -> function
  | Leaf -> Leaf
  | Node (t1, v, t2) -> Node (map f t1, f v, map f t2)




let _ = map (fun x -> x+1) good
```

Exemples réalistes : fonctions d'impression

- Le type de `Printf.printf` de la bibliothèque standard est :
 $('a, \text{out_channel}, \text{unit}) \text{ format} \rightarrow 'a$
- Le premier argument de cette fonction détermine le nombre des arguments suivants, et leurs types.
- Avec le *functional unparsing* d'Olivier Danvy, le format est une *combinaison* de fonctions d'affichage élémentaires.
- On peut donner un type précis au format et à la fonction d'impression :

```
type 'a ty =  
  | Int : int ty  
  | String : string ty  
  | Pair : 'a ty * 'b ty -> ('a * 'b) ty  
  
let rec printf : type a. a ty -> a -> unit =  
  function  
  | Int -> print_int (* ici a est int *)  
  | String -> print_string (* ici a est string *)  
  | Pair (b, c) -> (* ici a est une paire *)  
    fun (vb,vc) -> printf b vb; printf c vc  
  
let ( ++ ) = fun x y -> Pair (x,y)  
  
let _ = printf ((Int++String)++Int) ((4,"_then_",5))
```

Pour en savoir plus

-  Hongwei Xi, Chiyan Chen, and Gang Chen.
Guarded recursive datatype constructors.
[In Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages](#), pages 224–235, New Orleans, January 2003.
-  François Pottier and Yann Régis-Gianas.
Stratified type inference for generalized algebraic data types.
[In Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '06](#), pages 232–244, New York, NY, USA, 2006. ACM.
-  Dimitrios Vytiniotis, Simon L. Peyton Jones, Tom Schrijvers, and Martin Sulzmann.
Outsidein(x) modular type inference with local assumptions.
[J. Funct. Program.](#), 21(4-5) :333–412, 2011.