Programmation Fonctionnelle Avancée 10 : Types fantomes et GADT

Ralf Treinen

Université Paris Diderot
UFR Informatique
Institut de Recherche en Informatique Fondamentale

treinen@irif.fr

21 novembre 2018

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 regarde que le corps de la définition : `float`.

Exemples (phantom1.ml)

```
type 'a t = float;;

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

x+.y;;
```


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 (+.) = Pervasives.(+.)
  let to_float f = f
end;;
```


Exemples (phantom3.ml)

```
open Length

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

m1 +. m1;;
f1 +. f1;;
to_float (f1 +. f1);;
m1 +. f1;;
```


Même les meilleurs peuvent en avoir besoin

CNN.com

sci.tech> space> story page

exploringmars

in-depth specials

MAIN PAGE
WORLD
U.S.
LOCAL
POLITICS
WEATHER
BUSINESS
SPORTS
TECHNOLOGY
SPACE
HEALTH
ENTERTAINMENT
BOOKS
TRAVEL
FOOD
ARTS & STYLE
NATURE
IN-DEPTH
ANALYSIS
myCNN

Headline News brief
news quiz
daily almanac

MULTIMEDIA:
video
video archive
audio
multimedia showcase
more services

Metric mishap caused loss of NASA orbiter

September 30, 1999
Web posted at: 4:21 p.m. EDT (2021 GMT)

In this story:

Metric system used by NASA for many years

Error points to nation's conversion lag

RELATED STORIES, SITES

NASA's Climate Orbiter was lost September 23, 1999

By Robin Lloyd
CNN Interactive Senior Writer

(CNN) -- NASA lost a $125 million Mars orbiter because a Lockheed Martin engineering team used English units of measurement while the agency's team used the more conventional metric system for a key spacecraft operation, according to a review finding released Thursday.

The units mismatch prevented navigation information from transferring between the Mars Climate Orbiter

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 ('a, 'b) t
    val listevide : (vide, 'b) t
    val cons : 'b -> ('a, 'b) t -> (nonvide, 'b) t
    val head : (nonvide, 'b) t -> 'b
  end;;

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


Exemples (phantom5.ml)

```
open Liste;;

listevide;;
cons 3 listevide;;
head (cons 3 listevide);;

  (* erreur de typage ! *)
head listevide;;
```


Listes vides et non vides

- L'exemple précédent permet de distinguer les listes vides et non vides aux *niveau de typage*.
- Problème : quel est le type de la fonction tail ?
- Son argument doit être du type (nonvide,'b) 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.


```
module type LISTEVP =  sig
    type ('a, 'b) t
    val listevide : ([ `Vide ], 'b) t
    val cons : 'b -> ([< `Vide | `Nonvide ], 'b) t
                -> ([ `Nonvide ], 'b) t
    val head : ([ `Nonvide ], 'b) t -> 'b
    val tail : ([ `Nonvide ], 'b) t -> ([< `Vide | `Nonvide ]
  end;;

module ListeVP:LISTEVP = struct
  type ('a, 'b) t = 'b 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;;

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

tail (cons 1 listevide);;

tail (tail (cons 1 listevide));; (* assert failure *)
```


Pourquoi cet 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 vide* est trop grossière.
- Idée : coder la longueur de la liste dans le type !
- Pour faire cela nous devrons représenter chaque nombre naturel par un *type* différent : nous utilisons un type zero, et un constructeur de type 'a succ.
- Ainsi, le nombre naturel 2 est représenté par le type zero succ succ (notation postfix pour les applications de constructeurs de type).


```
module type LISTECOUNT =  sig
    type ('lon, 'ele) t
    type zero
    type 'a succ
    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
  type zero
  type 'a succ
  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 ;;
```


Exemples (phantom9.ml)

```
open Listecount;;

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

let l1 = tail (tail (tail l4));;

head l1;;

tail (tail l1);;
```

Listes avec codage de longueur

- Les deux derniers transparents poussent 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 : 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 `http://lablgtk.forge.ocamlcore.org/` et le message de Jacques Garrigue de septembre 2001
`http://caml.inria.fr/pub/ml-archives/caml-list/2001/09/2915ad47e671450ac5acefe4d8bd76fb.en.html`

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`

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 ();;
status conn_ro;;
destroy conn_ro;; (* error *)

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



```
module type HTML = sig
    type +'a elt
    val pcdata : string -> [> `Pcdata ] elt
    val span : [< `Span | `Pcdata ] elt list
               -> [> `Span ] elt
    val div : [< `Div | `Span | `Pcdata ] elt list
              -> [> `Div ] elt
  end
;;

module Html : HTML =
  struct
    type raw =
      | Node of string * raw list
      | Pcdata of string
    type 'a elt = raw
    let pcdata s = Pcdata s
    let div l = Node ("div",l)
    let span l = Node ("span",l)
  end;;
```


Exemples (html2.ml)

```
open Html;;
let x = div [ pcdata "" ];;

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

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


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.

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.

Exemples (gadt1.ml)

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

(* pattern matching non exhaustive *)
```


Vers des GADT

- De l'anglais : *Generalised Algebraic Data Types*
- Il s'agit d'une extension du langage (contrairement aux types fantômes).
- Première étape de l'extension : on sépare *le type envoyé* par les constructeurs de la *définition* du type : cela permet de préciser, et distinguer, le résultat de chaque constructeur.
- Syntaxe alternative (et plus générale) pour déclarer les constructeurs d'un type somme
- Attention : à partir d'ici nous avons besoin de OCaml > 4.00.

Syntaxe alternative des types algébriques

- La syntaxe vue en L3 :

```
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.

Exemples (gadt2.ml)

```
type empty
type nonempty
type 'a il =
  | Cons : int * 'a il -> nonempty il
  | Nil : empty il

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

(* mais le typeur exige que tous les motifs
   ont le meme type ! *)
let tail_or_empty = function
  | Nil -> 1
  | (Cons _) as l -> head l;;
```


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 differente de ce type.
 *)

let tail_or_empty : type a . a il -> int = function
  | Nil -> 1
  | (Cons _) as l -> head l;;
```


Exemples (gadt4.ml)

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

let silly : type a . a il -> int = function
  | Nil -> 1
  | (Cons _) as l ->
      (* here we know that 'a = nonempty! *)
       match l with Cons (x,_) -> 2;;
```

Exemples (gadt5.ml)

```
(* Exemple: les listes vides/nonvides, encore *)

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

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


Exemples (gadt7.ml)

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

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


Exemples réalistes : interpréteur (sans GADT)

- Interpréteur pour un petit langage d'expressions
- Types de base int, bool, plus un *constructeur de type* P.
- Nombre infini de types d'expressions (int, bool, P(int,bool), P(int,P(bool,bool)), ...)
- On veut avoir un seul type d'expressions

Exemples (gadt8.ml)

```
(* expressions sans GADT, a la L3 *)

type expr =
  | Int of int
  | Bool of bool
  | Prod 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
  | Prod (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 "Nonboolean condition in an ITE!";;

(* exception pendant l'execution *)
eval (IfThenElse ((Int 0),(Int 1),(Int 2)));;

(* OK *)
eval (IfThenElse ((Bool true),(Int 3),(Int 4)));;
eval (IfThenElse ((Bool true),(Bool false),(Bool true)));;
```


Limitations de l'interpréteur sans GADT

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

- aplatir les expressions dans un type expr
- aplatir les résultats dans un type res
- écrire la fonction eval en traitant les possibles cas d'erreur qui viennent du fait que toute expression peut apparaître partout

Exemples (gadt10.ml)

```
(* expression avec GADT *)

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


Exemples (gadt11.ml)

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

(* erreur de typage ! *)
eval (IfThenElse ((Int 0),(Int 1),(Int 2)));;
```

Exemples réalistes : fonctions composables

- Comment typer une liste [f1;f2;f3 ;...; fn] contenant des fonctions qui se composent ?

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

- Cela semble impossible avec le langage vu en L3.
- Avec GADT, on peut donner un type précis qui permet d'écrire du code bien typé.

Exemples (gadt12.ml)

```
type ('a,'b) cfl =
  | Nilf: ('a,'a) cfl
  | Consf: ('a -> 'b) * ('b,'c) cfl -> ('a,'c) cfl;;

let rec compute : type a b. a -> (a,b) cfl -> b
  = fun x ->
  function
  | Nilf -> x (* here 'a = 'b *)
  | Consf (f,rl) -> compute (f x) rl;;

let bad = Consf((fun x -> truncate x),
                Consf(string_of_float,Nilf));;
let good = Consf((fun x -> truncate x),
                 Consf(string_of_int,Nilf));;
compute 3.5 good;;
```


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 de 'b.
- Il s'agit de variables dites *existentielles*, parce que le type de Consf

$$\forall acb.(a \to b) * (b,c)\ cfl \to (a,c)\ cfl$$

peut se lire comme

$$\forall ac(\exists b.(a \to b) * (b,c)\ cfl) \to (a,c)\ cfl$$

Exemples réalistes : arbres binaires completes

Un type pour les arbres binaires completes : tous les chemins de la racine vers des feuilles ont la même longueur.

```
type ('a,'h) bintree =
  | Leaf: ('a,unit) bintree
  | Node: ('a,'h) bintree * 'a * ('a,'h) bintree -> ('a,'h*unit) b
;;

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) bintree -> (b,h) bintree
  = function f -> function
               | Leaf -> Leaf
               | Node(t1,v,t2) -> Node(map f t1, f v, map f t2)
;;

map (function x -> x+1) good;;
```

Exemples réalistes : fonctions d'impression

- Le type de Printf.printf de la bibliothèque standard est :

```
('a, out_channel, unit) format -> '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 = fun format v ->
 match format with
  | Int -> print_int v (* ici v est int *)
  | String -> print_string v (* ici v est string *)
  | Pair (b, c) -> printf b (fst v); printf c (snd v)
      (* ici v est une paire *)
;;

let ( ** ) = fun x y -> Pair (x,y);;

printf ((Int ** String) ** Int) ((4, " and 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.