Usage avancé du système de type

- Rappel sur le typage en OCaml
- Variants polymorphes
- Types fantômes
- Types algébriques généralisés

Rappel sur le typage en OCaml

Système de types *polymorphe* : List.map manipule des listes de tout type

```
List.map;;
|| - : ('a -> 'b) -> 'a list -> 'b list = <fun>

List.map (fun x -> x + 1) [1;2;3];;
|| - : int list = [2; 3; 4]

List.map (fun s -> s^"-") ["a";"b";"c"];;
|| - : string list = ["a-"; "b-"; "c-"]
```


Rappel sur le typage en OCaml

Inférence de types : le système découvre tout seul le type le plus général, sans besoin de déclarer les types

```
let k x y = x;;
|| val k : 'a -> 'b -> 'a = <fun>

let s x y z = (x z) (y z);;
|| val s : ('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c = <fun>
```


Les règles de typage pour le noyau de OCaml

Fun
$$\frac{A, x : t_2 \vdash e : t_1}{A \vdash \mathit{fun}\ x \text{ -> } e : t_2 \to t_1}$$

App
$$\frac{A \vdash e_1 : \tau_2 \to \tau_1 \qquad A \vdash e_2 : \tau_2}{A \vdash e_1\ e_2 : \tau_1}$$

Var-Inst
$$\frac{z : \forall \overline{\sigma}.\tau_0 \in A}{A \vdash z : \tau_0[\overline{\sigma} := \overline{\tau}]}$$

Gen-Let
$$\frac{A \vdash e_1 : \tau_1 \qquad A, x : \mathit{Gen}(\tau_1, A) \vdash e_2 : \tau_2}{A \vdash \mathit{let}\ x = e_1\ \mathit{in}\ e_2 : \tau_2}$$

where $\mathit{Gen}(\tau, A) = \forall \overline{\sigma}.\tau_1$ (for $\overline{\sigma} = FV(\tau) \setminus FV(A)$)

Quelques résultats fondamentaux

- il existe un algorithme qui, étant donné une expression e, trouve, si elle est typable, son type σ *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 Robinson que vous avez vu en Logique
- les algorithmes modernes utilisent plutôt directement de la rèsolution de contraintes
- à la surprise générale, en 1990 on a montré que l'inférence de type pour le noyau de ML est DEXPTIME complète.

Comment faire exploser le typeur d'OCaml...

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

```
let x0 = fun x -> x in
 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;;
```

Le type de `xn` a taille 2^n ! ! !

Remarque

Heureusement, en pratique, personne n'écrit de code comme ça. L'inférence de type à la ML reste un des système de type les plus puissants.

Les règles de typage pour OCaml : ...

Standard :

- sommes
- tuples
- enregistrements

Plus compliqué : *value restriction*

- typage des effets de bord

Très compliqué : *sous-typage*

- modules
- *recursion polymorphe*
- objets
- *variants polymorphes*
- *GADT*

Petit détour par la value restriction ...

En OCaml, on dispose de structures mutables capables de contenir des données de tout type.
Si on authorise leur usage polymorphe, le code suivant serait accepté :

```
let r: 'a option ref = ref None;;

let r1: string option ref = r;;
let r2: int option ref = r;;

r1 := Some "foo";;

let (Some v : int option) = !r2 in v+1;;
```

On a cassé le système de types : on a converti un `string` en `int` !
Il faut donc s'assurer que les structures mutables (comme les `ref`) ne puissent jamais contenir des *valeurs de types différents*.

Petit detour par la value restriction ...

Une solution très simple (utilisée par OCaml et SML) est de permettre la généralisation *seulement* pour les valeurs (d'où le nom).
Les valeurs sont :

- les constantes (13, "foo", 13.0, ...)
- les variables (x, y, ...)
- les fonctions (fun x -> e)
- les constructeurs appliqués à des valeurs (Foo v), excepté **ref**
- une valeur avec une contrainte de type (v : t)
- un n-uplet de valeurs (v1, v2, ...)
- un enregistrement contenant seulement des valeurs {l1 = v1, l2 = v2, ...}
- une liste de valeurs [v1, v2, ...]

Astuces pour la value restriction ...

Cette restriction peut rejeter des programmes valides :

```
let id x = x;;
| val id : 'a -> 'a = <fun>

let f : 'a -> 'a  = id id;;
| val f : '_a -> '_a = <fun>
```

le `'_a` dans le type signifie que la variable a sera instanciée une seule fois.
Il est souvent facile de corriger le problème, en faisant une η-expansion (transformer `f` en `fun x -> f x`) :

```
let f : 'a -> 'a  = fun x -> id id x;;
| val f : 'a -> 'a = <fun>
```


Relaxed value restriction

La version actuelle de OCaml utilise la *relaxed* value restriction, introduite par Jacques Garrique, basée sur l'idée qu'on peut généraliser les variables de type qui apparaissent seulement *positivement* dans un type (voir définition au tableau).

```
let l = [];;
| val l : 'a list = []

let f l l' = if l = l' then l else l@l';;
| val f : 'a list -> 'a list -> 'a list = <fun>

f l l;;
| - : 'a list = []

f l;;
| - : '_a list -> '_a list = <fun>
```


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.


David McAllester.
A logical algorithm for ML type inference.
In Proceedings of the 14th international conference on Rewriting techniques and applications, RTA'03, pages 436–451, Berlin, Heidelberg, 2003. Springer-Verlag.


Andrew K. Wright.
Simple imperative polymorphism.
Lisp Symb. Comput., 8(4) :343–355, December 1995.


Jacques Garrigue.

Les Variants Polymorphes

Rappel sur les types somme

On connait déjà les types sommes (appelés aussi *variants*), par ex. :

```
type num = Zero | S of num;;
|| type num = Zero | S of num

let rec add = function
| (Zero,n) -> n
| (S n, m) -> S (add (n,m));;
|| val add : num * num -> num = <fun>

let two = S ( S (Zero));;
|| val two : num = S (S Zero)

add (two,two);;
|| - : num = S (S (S (S Zero)))
```

Les constructeurs Zero et S ne peuvent pas être utilisés si le type num n'est pas précédemment défini.

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 somme ayant le même constructeur, seule la dernière déclaration est active

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 " ` ".

Un variant polymorphe :

- *peut appartenir à plusieurs types*
- peut *être utilisé avec plusieurs types d'arguments*
- donne lieu à des types qui sont des listes de variants, avec possiblement des bornes supérieures ($<$) et/ou inférieures ($>$), et une rélation de sous-typage qui est réalisée à travers le polymorphisme (d'où le nom).

Variants polymorphes : borne inférieure

Observez les types de ces *valeurs* :

```
`A;;
|| - : [> `A ] = `A

`B 4;;
|| - : [> `B of int ] = `B 4

`A, `B 3;;
|| - : [> `A ] * [> `B of int ] = (`A, `B 3)

let l = [`A ; (`B 4)];;
|| val l : [> `A | `B of int ] list = [`A; `B 4]
```

Le type de l *peut contenir* le constructeur `A sans argument et le constructeur `B avec un argument entier.

Variants polymorphes : borne inférieure

- le « > » signifie : « un type qui *peut contenir au moins* les constructeurs suivants » ; c'est une *borne inférieure* qui apparaît quand on *produit un résultat* dont on sait qu'il peut contenir au moins les constructeurs qui apparaîssent dans le type
- la barre verticale signifie « ou ».

Attention le type [> `A | `B of int] est *une instance*, à la fois de [> `A ] et [> `B of int]

Les incompatibilités

Dans une même fonction, on ne peut pas utiliser un variant avec des types différents :

```
let f = function
  | `A   -> 1
  | `A 3 -> 2;;
|| Characters 37-41:
||     | `A 3 -> 2;;
||       ^^^^
|| Error: This pattern matches values of type [? `A of int ]
||        but a pattern was expected which matches values of type [? `A ]
||        Types for tag `A are incompatible
```


Variants polymorphes : borne supérieure

Observez les types de cette *fonction* :

```
let f = function `B x -> x | `A x -> x;;
|| val f : [< `A of 'a | `B of 'a ] -> 'a = <fun>
```

- 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 qui apparaîssent dans le type

Expliquez le type de cette fonction :

```
let switch = function `A -> `B | `B -> `A;;
|| val switch : [< `A | `B ] -> [> `A | `B ] = <fun>
```

Le bornes supérieures et infèrieures ensemble

```
let f = function `S a -> 1 | `Z -> 2;;
|| val f : [< `S of 'a | `Z ] -> int = <fun>

let g = function `S _ -> 4 | _ -> 5;;
|| val g : [> `S of 'a ] -> int = <fun>

let h = function `S _ -> 4 | x -> f x;;
|| val h : [< `S of 'a | `Z > `S ] -> int = <fun>
```

- f a un $<$ parce-que il n'accepte que ces deux cas
- g a un $>$ parce-que grâce au cas _ -> 5 il accepte tout type qui peut contenir *au moins* `S
- h a un $>$ comme g, mais le x doit être acceptable pour f, d'où la même borne supérieure $<$ que f

On peut nommer les types

On peut utiliser les types des variants dans une définition de type

```
type colormix =
        Black
      | Composite of [`Red | `Blue | `Green] list;;

Black;;
|| - : colormix = Black

Composite [ `Blue; `Green];;
|| - : colormix = Composite [`Blue; `Green]
```


Attention aux types ouverts...

On ne peut pas utiliser les bornes directement dans les définitions : un type avec borne, aussi appelé *type ouvert* represente *un ensemble* de types (tous les types compris entre les bornes), et pas un seul type ; on ne peut donc pas les nommer :

```
type tf = [`A | `B of int];;
|| type tf = [ `A | `B of int ]

type bad = [> `A | `B of int];;
|| Characters 5-29:
||   type bad = [> `A | `B of int];;
||       ^^^^^^^^^^^^^^^^^^^^^^^^
|| Error: A type variable is unbound in this type declaration.
|| In type [> `A | `B of int ] as 'a the variable 'a is unbound
```


Attention aux types ouverts...

Par contre, on peut utiliser ces types ouverts dans des contraintes de type :

```
let f = function `B -> `A;;
|| val f : [< `B ] -> [> `A ] = <fun>

let f = function `B -> (`A : [> `A | `B]);;
|| val f : [< `B ] -> [> `A | `B ] = <fun>
```

Soustypage

Les types variant 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.

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

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

Le type 'a vlist est inclus dans le type 'a wlist, et il est donc un sous-type de 'a wlist.

Donc, dans les contextes ou on attend "au plus" les constructeurs de 'a wlist, on pourra utiliser tout aussi bien une 'a vlist.

Cas d'usage : constructeurs surchargès

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 sousensemble 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 *)
```


Soustypage et pattern matching

Considérons la fonction

```
let g = function `Nil -> 0
       | `Cons (_,_) -> 1 | `Snoc (_,_) -> 2;;
|| val g : [< `Cons of 'a * 'b | `Nil | `Snoc of 'c * 'd ] -> int = <fun>
```

à différence de ce qui se passe avec les types sommes, elle est applicable à plusieurs types différents (les sous-types de 'a wlist).

```
let vl = (`Nil : 'a vlist);;

let wl = (`Snoc(`Nil ,4) : 'a wlist);;

g wl;;
|| - : int = 2

g vl;;
|| - : int = 0
```


Coercion et Soustypage

On peut vouloir convertir une 'a vlist en 'a wlist ;
on ne sait plus 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). (On peut parfois omettre le : t1).
Exemple :

```
let wlist_of_vlist l = (l : 'a vlist :> 'a wlist);;
|| val wlist_of_vlist : 'a vlist -> 'a wlist = <fun>

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

wlist_of_vlist a;;
|| - : int wlist = `Cons (1, `Nil)
```

Coercion et Soustypage

Les coercions peuvent être utilisées un peu partout :

```
let open_vlist l = (l : 'a vlist :> [> 'a vlist]);;
|| val open_vlist : 'a vlist -> [> 'a vlist ] = <fun>

open_vlist vl;;
|| - : [> 'a vlist ] = `Nil


let switch (x: [`A | `B]) =
    (match x with `A -> `B | `B -> `A :> [`A | `B]);;
|| val switch : [ `A | `B ] -> [ `A | `B ] = <fun>
```


Covariance et contravariance

Comme les variants polymorphes introduisent du sous-typage, il faut savoir comment ce sous-typage se propage dans le reste des programmes.
Le cas le plus subtil est celui des fonctions : leur régles de sous-typage sont particulières.

Sub-Fun

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

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 -).

Un exemple des sous-typage de fonctions

```
(* the type of f is a subtype of g, and not the other way around *)
let f : [`T | `Z] -> int = function `T -> 1 | `Z -> 2;;
|| val f : [ `T | `Z ] -> int = <fun>

let g : [ `T] -> int = function `T -> 4 ;;
|| val g : [ `T ] -> int = <fun>

(f :> [ `T] -> int);;
|| - : [ `T ] -> int = <fun>

(g :> [ `T | `Z] -> int);;
|| Characters 0-24:
||   (g :> [ `T | `Z] -> int);;
||    ^^^^^^^^^^^^^^^^^^^^^^^^
|| Error: Type [ `T ] -> int is not a subtype of [ `T | `Z ] -> int
|| The second variant type does not allow tag(s) `Z
```

Donc on peut utiliser f a la place de g, mais pas le contraire

Un exemple des sous-typage de fonctions

```
(* this one expects a function able to handle T *)

let k f = function `S _ -> 4 | `T -> f `T;;
|| val k : ([> `T ] -> int) -> [< `S of 'a | `T ] -> int = <fun>

fun x -> k f x;;
|| - : [< `S of 'a | `T ] -> int = <fun>

fun x -> k g x;;
|| - : [< `S of 'a | `T ] -> int = <fun>
```

Un exemple des sous-typage de fonctions

```
(* this one expects a function able to handle Z or T *)

let h f = function `S _ -> 4 | `Z -> f `Z | `T -> f `T;;
|| val h : ([> `T | `Z ] -> int) -> [< `S of 'a | `T | `Z ] -> int = <fun>

fun x -> h f x;;
|| - : [< `S of 'a | `T | `Z ] -> int = <fun>

fun x -> h g x;;
|| Characters 11-12:
||   fun x -> h g x;;
||            ^
|| Error: This expression has type [ `T ] -> int
||        but an expression was expected of type [> `T | `Z ] -> int
||        The first variant type does not allow tag(s) `Z
```


Les annotations de variance sur les types abstraits

Comme vous le voyez, 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 connait, 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 t
type (-'a) tt
```


Cas d'usage : utilisation du soustypage

On peut encoder (partie d')une DTD avec des variants polymorphes, qui assurent qu'on ne construira que des documents HTML valides.

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


Cas d'usage : utilisation du soustypage

```
open HTML;;

let x = div [ pcdata "" ];;
|| val x : [> `Div ] HTML.elt = <abstr>

(* erreur, pas de div dans les span *)
let x' = span [ div [] ];;
|| Characters 55-61:
||   let x' = span [ div [] ];;
||                   ^^^^^^
|| Error: This expression has type [> `Div ] HTML.elt
||        but an expression was expected of type
||          [< `Pcdata of int | `Span ] HTML.elt
||        The second variant type does not allow tag(s) `Div

let x'' = span [ span [] ];;
|| val x'' : [> `Span ] HTML.elt = <abstr>
```

Cas d'usage : utilisation du soustypage

```
let f s = div [ s; pcdata ""];;
|| val f :
||   [< `Div | `Pcdata of int | `Span > `Pcdata ] HTML.elt -> [> `Div ] HTM
|| L.elt =
||   <fun>

let f' s = div [ s; span []];;
|| val f' :
||   [< `Div | `Pcdata of int | `Span > `Span ] HTML.elt -> [> `Div ] HTML.
|| elt =
||   <fun>

let f'' s = div [ s; span []; pcdata "" ];;
|| val f'' :
||   [< `Div | `Pcdata of int | `Span > `Pcdata `Span ] HTML.elt ->
||   [> `Div ] HTML.elt = <fun>
```


Contraintes dans les types

Attention en jouant avec des variants polymorphes, on peut se retrouver confronté à des types inhabituels

```
let f1 = fun (`Nombre x) -> x = 0;;
|| val f1 : [< `Nombre of int ] -> bool = <fun>

let f2 = fun (`Nombre x) -> x = 0.0;;
|| val f2 : [< `Nombre of float ] -> bool = <fun>

let f x = f1 x || f2 x;;
|| val f : [< `Nombre of float & int ] -> bool = <fun>
```

La définition de f force le type de l'argument du constructeur `Nombre a être un int et un float à la fois (int & float). Dans cet exemple, cette contrainte est impossible à satisfaire. f est donc bien typéee, mais on ne pourra jamais l'appliquer.

Caveat emptor

Les variants polymorphes permettent d'écrire des programmes plus flexibles que les types sommes habituels. Mais ils ont leurs inconvénients :

- ils induisent des problèmes de typage parfois *complexes*
- moins de vérifications sont faites statiquement, et il devient facile d'utiliser un constructeur qui n'existe pas, ou avec un mauvais type, sans que le compilateur ne dise rien
- ils induisent une petite perte d'efficacité des programmes.

Donc n'utilisez les variants polymorphes que si vous en avez *vraiment* besoin.

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.

Les types fantômes

Les types fantômes (phantom types)

Un type fantôme est un type paramétrique dont au moins un paramètre 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 :

```
let (x: char t) = 3.0;;
|| val x : char t = 3.

let (y: string t) = 5.0;;
|| val y : string t = 5.

x+.y;;
|| - : float = 8.
```


Les types fantômes (phantom types)

En effet, les égalités char t = float = string t sont connues du typeur, qui ne regarde que le corps de la définition : float.

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.

Distinguer des unités de mésure différentes

On cache la définition de 'a t dans l'interface

```
module 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 = struct
  type 'a t = float
  let meters f = f
  let feet f = f
  let (+.) = (+.)
  let to_float f = f
end;;
```

Le typeur distingue les mètres des pieds

```
open Length
open Printf
let m1 = meters 10.;;
| val m1 : [ `Meters ] Length.t = <abstr>

let f1 = feet 40.;;
| val f1 : [ `Feet ] Length.t = <abstr>

printf "10m␣+␣10m␣=␣%g\\n" (to_float (m1 +. m1));;
| 10m + 10m = 20
| - : unit = ()

printf "40ft␣+␣40ft␣=␣%g\\n" (to_float (f1 +. f1));;
| 40ft + 40ft = 80
| - : unit = ()

m1 +. f1;;
| Characters 6-8:
|   m1 +. f1;;
|        ^^
| Error: This expression has type [ `Feet ] Length.t
|        but an expression was expected of type [ `Meters ] Length.t
|        These two variant types have no intersection
```

Il n'est pas nécéssaire d'utiliser des variants, mais c'est pratique.

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

E-MAIL:
Subscribe to one of our news e-mail lists.

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 spacecraft team in at Lockheed Martin in Denver and the flight team at NASA's Jet Propulsion Laboratory in

Les listes vides et les listes non vides

```
module 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 : 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);;
```


En pratique :

```
open Liste;;

listevide;;
| - : (Liste.vide, 'b) Liste.t = <abstr>

cons 3 listevide;;
| - : (Liste.nonvide, int) Liste.t = <abstr>

head (cons 3 listevide);;
| - : int = 3

head listevide;;
| Characters 5-14:
|   head listevide;;
|        ^^^^^^^^^
| Error: This expression has type (Liste.vide, 'a) Liste.t
|        but an expression was expected of type (Liste.nonvide, 'b) Liste
| t
```

Types fantômes avec variants polymorphes

Si on utilise des variants polymorphes comme paramètres, le sous-typage des variants polymorphes donne des types plus précis pour les constructeurs :

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


En pratique :

```
open ListeVP;;

listevide;;
|| - : ([ `Vide ], 'b) ListeVP.t = <abstr>

cons 3 listevide;;
|| - : ([ `Nonvide ], int) ListeVP.t = <abstr>

head (cons 3 listevide);;
|| - : int = 3

head listevide;;
|| Characters 5-14:
||   head listevide;;
||        ^^^^^^^^^
|| Error: This expression has type ([ `Vide ], 'a) ListeVP.t
||        but an expression was expected of type ([ `Nonvide ], 'b) ListeV
|| .t
||        These two variant types have no intersection
```


On pousse l'exercice : coder la longueur de la liste

```
module 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 : sig
  type ('a, 'b) t
  type zero
  type 'a succ
  val listevide : (zero, 'b) t
  val estvide : ('a, 'b) t -> bool
  val cons : 'b -> ('a, 'b) t -> ('a succ, 'b) t
  val head : ('a succ, 'b) t -> 'b
  val tail : ('a succ, 'b) t -> ('a, 'b) t
end);;
```


En pratique :

```
open Listecount;;

let l4 = cons 1 (cons 2 (cons 3 (cons 4 listevide)));;
|| val l4 :
||   (Listecount.zero Listecount.succ Listecount.succ Listecount.succ
||    Listecount.succ, int)
||   Listecount.t = <abstr>

let l1 = tail (tail (tail l4));;
|| val l1 : (Listecount.zero Listecount.succ, int) Listecount.t = <abstr>

head l1;;
|| - : int = 4

tail (tail l1);;
|| Characters 5-14:
||   tail (tail l1);;
||        ^^^^^^^^^
|| Error: This expression has type (Listecount.zero, int) Listecount.t
||        but an expression was expected of type
||          ('a Listecount.succ, 'b) Listecount.t
```

Mais l'utilité est limitée : on ne sait pas trop écrire map avec ces types de données (essayez !).

Cas d'usage : HTML bien typé

On peut encoder (partie d')une DTD avec des variants polymorphes, qui assurent qu'on ne construira que des documents HTML valides.

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


Cas d'usage : 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'usage : contrôle d'accès dans `libvirt`

Voir :
`http://camltastic.blogspot.fr/2008/05/phantom-types.html`

```
module 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 = 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;;
```


Cas d'usage : contrôle d'accès dans `libvirt`

```
open Connection;;

open Printf;;

let conn = connect_readonly () in
printf "status = %d\n" (status conn);;
|| status = 1
|| - : unit = ()

(* destroy conn;; (* error *) *)
let conn = connect () in
printf "status = %d\n" (status conn);
destroy conn;;
|| status = 2
|| - : unit = ()
```

Bilan

avantages :

- *étiquetes* sur les types qui permettent
- un contrôle *statique* du bon usage des données
- sans aucune penalité 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.

Les Types algébriques généralisés, ou GADT

Attention : ce qui suit nécessite OCaml v 4.00 ou suivant

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 exhaustif... et pourtant :

```
let silly x = match x with
  [] -> 1
| a::r -> match x with (a'::r') -> 2;;
|| Characters 47-73:
||     | a::r -> match x with (a'::r') -> 2;;
||                ^^^^^^^^^^^^^^^^^^^^^^^^^^
|| Warning 8: this pattern-matching is not exhaustive.
|| Here is an example of a value that is not matched:
|| []
|| val silly : 'a list -> int = <fun>
```

On aimerait pouvoir obtenir de l'information sur le type des valeurs, à partir des tests faits à l'éxécution.

GADT, premiers pas

On peut séparer *le type retourné* par les constructeurs de leur *définition* : cela permet de préciser, et distinguer, le résultat de chaque constructeur :

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

Utilisation...

```
(* jusqu'ici, tout va bien *)
let hd = function Cons(x,r) -> x;;
|| val hd : nonempty il -> int = <fun>
```


GADT : annotations de polymorphisme

```
(* mais le typeur assume que tous les cas ont le meme type!*)
let hd' = function
  | Nil -> 1
  | (Cons _) as l -> hd l;;
|| Characters 98-106:
||     | (Cons _) as l -> hd l;;
||       ^^^^^^^^
|| Error: This pattern matches values of type nonempty il
||        but a pattern was expected which matches values of type empty i
```

On peut aider le typeur, et lui dire que hd' est une fonction ayant un *type polymorphe*, et donc chaque cas peut avoir une instance *différente* ce de type :

```
let hd' : type a . a il -> int = function
  | Nil -> 1
  | (Cons _) as l -> hd l;;
|| val hd' : 'a il -> int = <fun>
```


GADT, intéraction filtrage / typage

On peut enfin écrire une version de silly qui ne donne plus de warning :

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


Exemple : les listes vides/nonvides, encore

```
type empty
type nonempty
type ('a, 'b) l =
    Nil : (empty, 'b) l
  | Cons : 'b * ('a, 'b) l -> (nonempty, 'b) l;;

let hd = function Cons (x,r) -> x;;
|| val hd : (nonempty, 'a) l -> 'a = <fun>
```

Exemple : les listes vides/nonvides, encore

Mais cette fois, on peut les utiliser !

```
let rec length : type a b. (a,b) l -> int =
 function
  | Nil -> 0 | (Cons (_,r)) -> 1+ (length r);;
|| val length : ('a, 'b) l -> int = <fun>

let rec map :
     type a b c. (b -> c) -> (a,b) l -> (a,c) l
= fun f ->
   function
     | Nil -> Nil
     | (Cons (x,r)) -> Cons(f x,map f r);;
|| val map : ('b -> 'c) -> ('a, 'b) l -> ('a, 'c) l = <fun>
```


Exemple : les listes avec leur longeur, encore

On peut refaire les listes avec la longueur encodée dans le type

```
type zero
type 'n succ
type (_, 'a) l =
| Nil : (zero, 'a) l
| Cons : 'a * ('n, 'a) l -> ('n succ, 'a) l;;

let head : type a n . (n succ , a) l -> a =
  fun (Cons(x,_)) -> x;;
|| val head : ('n succ, 'a) l -> 'a = <fun>

let tail : type a n . (n succ , a) l -> (n, a) l =
  fun (Cons(_,r)) -> r;;
|| val tail : ('n succ, 'a) l -> ('n, 'a) l = <fun>
```


Exemple : les listes avec leur longeur, encore

Cette fois, à différence de ce qui se passait avec les types phantômes, la fonction `map` s'écrit sans difficulté :

```
let rec map :
    type a b n. (a -> b) -> (n, a) l -> (n, b) l =
  fun f ->
   function
     Nil -> Nil
   | Cons (x,r) -> Cons(f x, map f r);;
|| val map : ('a -> 'b) -> ('n, 'a) l -> ('n, 'b) l = <fun>
```

Par contre la fonction `fold` posera problème.

Remarque : pas de types à l'éxécution

Comme d'habitude en OCaml, le code produit par le compilateur ne contient pas d'information de type. Vous pouvez le vérifier en demandant qu'on vous montre la représentation intermédiaire :

```
ocamlopt -dlambda gadt-nlist.ml

(let (head/1013 (function param/1042 (field 0 param/1042)))
...
(let (tail/1020 (function param/1045 (field 1 param/1045)))
...
(letrec
  (map/1027
     (function f/1032 param/1048
       (if param/1048
         (makeblock 0 (apply f/1032 (field 0 param/1048))
           (apply map/1027 f/1032 (field 1 param/1048)))
         0a)))
```

Exemples réalistes : interpreteur (sans GADT)

Interprétation d'un petit langage d'expressions :

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

type res = I of int | B of bool | P of res * res;;

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!";;
|| val eval : expr -> res = <fun>

eval (IfThenElse ((Int 0),(Int 1),(Int 2)));;
|| Exception: Failure "Nonboolean condition in an ITE!".
```


Limitations de l'interpreteur sans GADT

Comme on ne sait pas classifier les expressions par leur type, on est obligés 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 réalistes : interpreteur (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;;

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);;
|| val eval : 'a expr -> 'a = <fun>
```


Exemples réalistes : interpreteur (avec GADT)

```
eval (IfThenElse ((Int 0),(Int 3),(Int 4)));;
|| Characters 18-25:
||   eval (IfThenElse ((Int 0),(Int 3),(Int 4)));;
||                     ^^^^^^^
|| Error: This expression has type int expr
||        but an expression was expected of type bool expr

eval (IfThenElse ((Bool true),(Int 3),(Int 4)));;
|| - : int = 3

eval (IfThenElse ((Bool true),(Bool false),(Bool true)));;
|| - : bool = false
```

Exemples réalistes : fonctions composables (sans GADT)

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

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

Sans GADT, on doit percer des trous dans le système de type :

```
type ('a,'b) cfl = ('a -> 'b) list;;
|| type ('a, 'b) cfl = ('a -> 'b) list

let mkone (f:'a -> 'b) = [f];;
|| val mkone : ('a -> 'b) -> ('a -> 'b) list = <fun>

let add (f: 'a -> 'b)  (fl : ('b,'c) cfl) : ('a,'c) cfl =
 (Obj.magic f):: (Obj.magic fl);;
|| val add : ('a -> 'b) -> ('b, 'c) cfl -> ('a, 'c) cfl = <fun>
```


Exemples réalistes : fonctions composables (avec GADT)

Avec GADT, on peut donner un type précis qui permet d'écrire du code bien typé :

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

Attention 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 : fonctions composables (avec GADT)

```
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;;
|| val compute : 'a -> ('a, 'b) cfl -> 'b = <fun>

let cl = Consf ((fun x -> Printf.sprintf "%d" x), Nilf);;
|| val cl : (int, string) cfl = Consf (<fun>, Nilf)

let cl' = Consf ((fun x -> truncate x), cl);;
|| val cl' : (float, string) cfl = Consf (<fun>, Consf (<fun>, Nilf))

compute 3.5 cl';;
|| - : string = "3"
```


Exemples réalistes : fonctions d'impression (sans GADT)

Avec le *functional unparsing* d'Olivier Danvy, le format est une *combinaison* de fonctions d'affichage élémentaires.

```
let int x = print_int x;;
|| val int : int -> unit = <fun>

let printf format v = format v;;
|| val printf : ('a -> 'b) -> 'a -> 'b = <fun>

printf int 3;;
|| 3- : unit = ()

let ( ** ) f1 f2 = fun (x,y) -> f1 x; f2 y;;
|| val ( ** ) : ('a -> 'b) -> ('c -> 'd) -> 'a * 'c -> 'd = <fun>

printf ((int ** int) ** int) ((4, 3),5);;
|| 435- : unit = ()
```

Exemples réalistes : fonctions d'impression (avec GADT)

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 *
;;
|| val printf : 'a ty -> 'a -> unit = <fun>


let ( ** ) = fun x y -> Pair (x,y);;
|| val ( ** ) : 'a ty -> 'b ty -> ('a * 'b) ty = <fun>

printf ((Int ** String) ** Int) ((4, " and then "),5);;
|| 4 and then 5- : unit = ()
```

Programmation Fonctionnelle Avancée
└Types algébriques généralisés

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.