Le caractère ` à la rescousse

Factorisation et réutilisation de code grâce aux variants polymorphes

Boris Yakobowski

INRIA Rocquencourt

12 février 2008

Les types algébriques en ML

Les types algébriques :

```
type arbre = Feuille of int | Noeud of arbre * arbre
let rec hauteur = function
  | Feuille -> 1
  | Noeud (a1, a2) -> max (hauteur a1) (hauteur a2) + 1
```

- présents dans tous les dialectes de ML ;
- extrêmement puissants et utiles ;
- permettent d'écrire du code très concis, tout en gardant des garanties statiques très fortes
- manquent cruellement dans d'autres langages "modernes" (Java) ; ajoutés dans Tom et Scala

Le typage des types algébriques

- Un type algébrique est isomorphe à un *type somme*

```
type t = Gauche of t | Droite of u
``Gauche : t -> t + u''
``Droite : u -> t + u''
```

Le typage des types algébriques

- Un type algébrique est isomorphe à un *type somme*

```
type t = Gauche of t | Droite of u
‘‘Gauche : t -> t + u’’
‘‘Droite : u -> t + u’’
```

- En ML, les types algébriques doivent êtres déclarés, et un constructeur ne peut appartenir qu'à un seul type
  - Simplification du typage

    `let v = Some 1`

    Le compilateur sait que v a pour type `int option`, et non `int + ?` pour un “?” restant à définir

Le typage des types algébriques

- Un type algébrique est isomorphe à un *type somme*

```
type t = Gauche of t | Droite of u
‘‘Gauche : t -> t + u’’
‘‘Droite : u -> t + u’’
```

- En ML, les types algébriques doivent êtres déclarés, et un constructeur ne peut appartenir qu'à un seul type
  - Simplification du typage
  - Les annotations de types (sur les filtrages) deviennent inutiles

```
let f (v : u+t) = match v with
  | Gauche t -> ...
  | Droite u -> ...
```

Le typage des types algébriques

- Un type algébrique est isomorphe à un *type somme*

```
type t = Gauche of t | Droite of u
‘‘Gauche : t -> t + u’’
‘‘Droite : u -> t + u’’
```

- En ML, les types algébriques doivent êtres déclarés, et un constructeur ne peut appartenir qu'à un seul type
  - Simplification du typage
  - Les annotations de types (sur les filtrages) deviennent inutiles
  - Le compilateur peut prévenir lorsqu'un filtrage est non exhaustif

Un compromis perdant un peu d'expressivité

- On ne peut pas combiner des types algébriques entre eux

```
type metal_pur = Argent | Plomb | Cuivre | Or
type alliage = Acier | Fonte | Laiton
let masse_volumique (m : metal_pur + alliage) = [...]
```

Un compromis perdant un peu d'expressivité

- On ne peut pas combiner des types algébriques entre eux

```
type metal_pur = Argent | Plomb | Cuivre | Or
type alliage = Acier | Fonte | Laiton
let masse_volumique (m : metal_pur + alliage) = [...]
```

- On ne peut pas raffiner un type inductif préalablement défini

```
let transmutation = function
  | Plomb | Or -> Or
  | Cuivre -> Cuivre
  | Argent -> Argent
```

Ici, on pourrait vouloir exprimer que le type résultant ne contient jamais `Plomb`

Des valeurs qui préexistent

- Les valeurs des variants polymorphes *préexistent*, au même titre que `1` ou `"Bonjour"`

```
# let argent = `Argent;;
val argent : [> `Argent ] = `Argent
```

Le ` sépare les variants algébriques des variants polymorphes

Des valeurs qui préexistent

- Les valeurs des variants polymorphes *préexistent*, au même titre que `1` ou `"Bonjour"`

```
# let argent = `Argent;;
val argent : [> `Argent ] = `Argent
```

- On peut les combiner entre eux

```
# `Argent :: `Or :: [];;
- : [> `Argent | `Or ] list = [`Argent; `Or]
```

Des valeurs qui préexistent

- Les valeurs des variants polymorphes *préexistent*, au même titre que `1` ou `"Bonjour"`

```
# let argent = `Argent;;
val argent : [> `Argent ] = `Argent
```

- On peut les combiner entre eux

```
# `Argent :: `Or :: [];;
- : [> `Argent | `Or ] list = [`Argent; `Or]
```

- Mais pas avec d'autres valeurs

```
# `Argent :: None :: [];;
This expression has type 'a option but is here used
with type [> `Argent ]
```

Quelques valeurs

```
# let a = `A and b1 = `B (1, true) and b2 = `B "foo"
val a : [> `A ] = `A
val b1 : [> `B of int  bool ] = `B (1, true)
val b2 : [> `B of string ] = `B "foo"
```

On a défini trois valeurs, a, b1 et b2.

Quelques valeurs

```
# let a = `A and b1 = `B (1, true) and b2 = `B "foo"
val a : [> `A ] = `A
val b1 : [> `B of int  bool ] = `B (1, true)
val b2 : [> `B of string ] = `B "foo"
```

On a défini trois valeurs, `a`, `b1` et `b2`.

On notera que `b1` et `b2` utilisent tous les deux le constructeur `` `B ``, avec deux arguments différents

(un peu comme s'il était défini par `type 'a t = `B of 'a`)

Une liste

Définissons une liste contenant plusieurs variants :

```
# let l1 = [ `A ; `B 1 ; `C false ] ; ;
val l1 : [> `A | `B of int | `C of bool ] list
```

Le type `[> `A | `B of int | `C of bool]` se lit “un type comprenant *au moins* les constructeurs `A, `B et `C, `B prenant un argument de type int et `C de type bool”.

Une liste

Définissons une liste contenant plusieurs variants :

```
# let l1 = [ `A ; `B 1 ; `C false ] ; ;
val l1 : [> `A | `B of int | `C of bool ] list
```

Le type `[> `A | `B of int | `C of bool]` se lit “un type comprenant *au moins* les constructeurs `A, `B et `C, `B prenant un argument de type int et `C de type bool”.

Le caractère > indique le ”au moins”, et permet d'étendre le type donné à l1.

Des types extensibles

```
# let l2 = `A :: `D :: l1 ; ;
val l2 : [> `A | `B of int | `C of bool | `D ] list =
                              [ `A ; `D ; `A ; `B 1 ; `C false ]
```

Le type de l2 reflète le fait qu'elle contient aussi le constructeur `D (sans argument).

Des types extensibles

```
# let l2 = `A :: `D :: l1 ; ;
val l2 : [> `A | `B of int | `C of bool | `D ] list =
                              [ `A ; `D ; `A ; `B 1 ; `C false ]
```

Le type de l2 reflète le fait qu'elle contient aussi le constructeur `D (sans argument).

```
let _ = `B 'b' :: l1 ; ;
This expression has type [> `A | `B of int | `C of bool ]
list but is here used with type [> `B of char ] list
Types for tag `B are incompatible
```

On ne peut pas donner des arguments différents au même constructeur.

Des types non instantiables

On limite le type de ‘A à ‘A lui-même

```
# let l = (‘A : [‘A]) :: []
val l : [ ‘A ] list = [‘A]
```

Des types non instantiables

On limite le type de ‘A à ‘A lui-même

```
# let l = (‘A : [‘A]) :: []
val l : [ ‘A ] list = [‘A]
```

On ne peut plus rajouter d'éléments autres que ‘A

```
# let l' = ‘B :: l ; ;
This expression has type [ ‘A ] list but is here used
with type [> ‘B ] list.
The first variant type does not allow tag(s) ‘B
```

Extension par sous-typage

Le type `[‘A]` est un *sous-type* du type `[‘A | ‘B]`.

On peut donc étendre le type de `l` *explicitement*, par une coercion

```
# let l' = (l :> [‘A | ‘B] list) ; ;
val l' : [ ‘A | ‘B ] list = [‘A]
```

Extension par sous-typage

Le type `[‘A]` est un *sous-type* du type `[‘A | ‘B]`.

On peut donc étendre le type de `l` *explicitement*, par une coercion

```
# let l' = (l :> [‘A | ‘B] list) ; ;
val l' : [ ‘A | ‘B ] list = [‘A]
```

Il est maintenant possible de rajouter ‘B

```
# ‘B :: l' ; ;
- : [ ‘A | ‘B ] list = [‘B ; ‘A]
```

Des fonctions

```
# let f = function
  | `A  → `B
  | `C | `D  → `D ; ;
val f : [< `A | `C | `D ]  → [> `B | `D ] = <fun>
```

Même écriture que sur les types algébriques usuels

Des fonctions

```
# let f = function
  | `A  → `B
  | `C | `D  → `D ; ;
val f : [< `A | `C | `D ]  → [> `B | `D ] = <fun>
```

Même écriture que sur les types algébriques usuels

La construction < est duale de la construction > : on peut réduire le domaine de `f`.

```
# let f ' = (f : ([ `A | `C ]  → [ `B | `D | `E ])) ; ;
val f ' : [ `A | `C ]  → [ `B | `D | `E ] = <fun>
```

Les modules de OCaml

Un module est :

- Un module de base “`struct ... end`” contenant une suite de déclarations
- Un foncteur prenant en argument une signature de module, et renvoyant un module “`functor (X : S) -> M`”

Une déclaration de module est :

- une déclaration de sous-module ou de signature de module “`module M = ...` ” ou “`module type S = ...` ”
- une déclaration de valeur “`let t = ...`”, de type, d’exception...

Grammaire des signatures de modules

On se limite aux signatures pour rester simple.

S	$::=$	$\mathit{sig}\ \overline{D}\ \mathit{end}$	Module de base
	$\mid$	$\mathit{functor}(X : S) \to S$	Foncteur
D	$::=$	$\texttt{module}\ X = S$	Déclaration de module
	$\mid$	$\texttt{module type}\ X = S$	Déclaration de signature
	$\mid$	d	Autre déclaration
d	$::=$	$\texttt{type}\ t = \ldots$	Déclaration de type
	$\mid$	$\texttt{val}\ v = \ldots$	Déclaration de valeur
	$\mid$	$\texttt{exception}\ e = \ldots$	Déclaration d'exception

Avec des variants polymorphes

```
type base_decl = [
  | `TypeDecl of name * type_decl
  | `ValDecl of name * val_decl
  | `ExceptionDecl of name * exception_decl
]

type mod_decl = [
  | `Struct of mod_item list
  | `Functor of name * mod_decl * mod_decl
] and mod_item = [
  | `Module of name * mod_decl
  | `Sig of name * mod_decl
  | base_decl
]
```

Travailler sur ces modules

Les modules de OCaml sont très généraux.

Deux simplifications possibles :

Travailler sur ces modules

Les modules de OCaml sont très généraux.

Deux simplifications possibles :

- Interdire les foncteurs prenant des foncteurs en argument

  À la place, on utilise un module intermédiaire contenant un foncteur (comme en SML).

Travailler sur ces modules

Les modules de OCaml sont très généraux.

Deux simplifications possibles :

- Interdire les foncteurs prenant des foncteurs en argument

  À la place, on utilise un module intermédiaire contenant un foncteur (comme en SML).
- Interdire les modules contenant des sous-modules de base (qui ne servent qu'a hiérarchiser le code)

Travailler sur ces modules

Les modules de OCaml sont très généraux.

Deux simplifications possibles :

- Interdire les foncteurs prenant des foncteurs en argument

  À la place, on utilise un module intermédiaire contenant un foncteur (comme en SML).
- Interdire les modules contenant des sous-modules de base (qui ne servent qu'a hiérarchiser le code)

On peut définir une transformation source $\longrightarrow$ source implémentant ces deux transformations.

Le langage de modules réduit

```
type mod_decl_r = [
    (* plus de foncteurs *)
  | `Struct of mod_item_r list
]

and mod_item_r = [
  |  `Module of name *
       [`Functor of name * mod_decl_r * mod_decl_r ]
        (* les modules de base ne sont plus autorisés *)
  | `Sig of name * mod_decl_r
  | base_decl
]
```

Les fonctions de conversion

```
let rec convert_mod : mod_decl -> mod_decl_r = function
  | `Struct l -> `Struct (convert_item_list l)
  | `Functor _ as f ->
      let name' = fresh_name () in
      `Struct [`Module (name', convert_functor f) ]

and convert_functor (`Functor (name, m1, m2)) =
  `Functor (name, convert_mod m1, convert_mod m2)
```

Les fonctions de conversion

```
let rec convert_mod : mod_decl -> mod_decl_r = function
  | `Struct l -> `Struct (convert_item_list l)
  | `Functor _ as f ->
      let name' = fresh_name () in
      `Struct [`Module (name', convert_functor f) ]

and convert_functor (`Functor (name, m1, m2)) =
  `Functor (name, convert_mod m1, convert_mod m2)

and convert_item = function
  | #base_decl as b -> [b]
  | `Module (name, `Struct l) -> convert_item_list l
  | `Module (name, (`Functor _ as f)) ->
      [`Module (name, convert_functor f)]
  | `Sig (name, m) -> [`Sig (name, convert_mod m)]
```

Les annotations, une aide pour le programmeur

```
and convert_functor (‘Functor (name, m1, m2)) =
  ‘Functor (name, convert_mod m1, convert_mod m2)
```

Le type inféré pour `convert_functor` est

```
val convert_functor :
  [ ‘Functor of name * mod_decl * mod_decl ] ->
  [ ‘Functor of name * mod_decl_r * mod_decl_r ]
```

Les annotations, une aide pour le programmeur

```
and convert_functor (‘Functor (name, m1, m2)) =
  ‘Functor (name, convert_mod m1, convert_mod m2)
```

Le type inféré pour `convert_functor` est

```
val convert_functor :
  [ ‘Functor of name * mod_decl * mod_decl ] ->
  [ ‘Functor of name * mod_decl_r * mod_decl_r ]
```

- Les annotations sont toujours facultatives
- Sans annotation, aucun des types inférés ne rentre sur une page
- Le compilateur signale les filtrages non exhaustifs

Les annotations, une aide pour le programmeur (2)

Et en cas d'erreur ?

```
and convert_item : mod_item -> mod_item_r list = function
  | `Module (name, `Struct l) -> l
  | [...]
```

Les annotations, une aide pour le programmeur (2)

Et en cas d'erreur ?

```
and convert_item : mod_item -> mod_item_r list = function
  | `Module (name, `Struct l) -> l
  | [...]

This expression has type mod_item list but is here used with
  type mod_item_r list

Type mod_item =  [...]
is not compatible with type mod_item_r =  [...]

Type mod_decl = [ `Functor of name * mod_decl * mod_decl
  | `Struct of mod_item list ] is not compatible with type
  mod_decl_r = [ `Struct of mod_item_r list ]
The second variant type does not allow tag(s) `Functor
```

Convertir un module restreint en un module général

L'ensemble des modules réduits est un sous-ensemble des modules généraux.

Convertir un module restreint en un module général

L'ensemble des modules réduits est un sous-ensemble des modules généraux.

La même relation existe sur les types :

```
let coerce_mod v = (v : mod_decl_r :> mod_decl)
let coerced_mod_item d = (d : mod_item_r :> mod_item)

val coerce_mod : mod_decl_r -> mod_decl
val coerced_mod_item : mod_item_r -> mod_item
```

Cette coercion est purement logique, elle n'a *aucun coût* à l'exécution.

Factorisation de code

- On se donne deux langages partageant certaines constructions, mais ayant chacun sa spécificité
- *But :* écrire un afficheur pour chaque langage

Factorisation de code

- On se donne deux langages partageant certaines constructions, mais ayant chacun sa spécificité
- *But :* écrire un afficheur pour chaque langage
  - Partage de code
  - Mais pas trop

Les deux langages

$$
\begin{array}{lcl}
 & & \textit{Source} \\
e & ::= & x \\
 & | & e\ \overline{e} \\
 & | & \lambda(\overline{x})\ e \\
 & | & \mathsf{let\ rec}^{?}\ x = e\ \mathsf{in}\ e \\
 & | & (e, ..., e) \\
 & | & e + e \mid e = e \mid \ldots
\end{array}
\qquad
\begin{array}{lcl}
 & & \textit{Noyau} \\
\epsilon & ::= & x \\
 & | & \epsilon\ \epsilon \\
 & | & \lambda(x)\ \epsilon \\
 & | & \mathsf{let}\ \ x = \epsilon\ \mathsf{in}\ \epsilon \\
 & | & (\epsilon, ..., \epsilon) \\
 & | & c
\end{array}
$$

Constructions partagées

- Les n-uplets, les variables et les constructionts `let` sont les mêmes dans les deux langages
- Les abstractions et applications sont multiples dans le langage source, et unaires dans le langage cible

Paramamétrer les types par le type des expressions

```
type 'expr common = [
  | `Var of var
  | `Uple of 'expr list
  | `Let of (var * 'expr) * 'expr ]

type 'expr source_aux = [
  | 'expr common
  | `SeqApp of 'expr * 'expr list
  | `SeqAbs of var list * 'expr
  | `Plus of 'expr * 'expr | `Eq of 'expr * 'expr ]

type 'expr dest_aux = [
  | 'expr common
  | `App of 'expr * 'expr | `Abs of var * 'expr
  | `Ct of ct ]
```

Fermer la récursion

```
type source = [ source source_aux ]
type dest = [ dest dest aux ]
```

Réponse du typeur :

```
type dest = [
  | `Var of var
  | `Uple of dest list
  | `Let of (var * dest) * dest
  | `App of dest * dest
  | `Abs of var * dest
  | `Ct of ct ]
```

Fermer la récursion

```
type source = [ source source_aux ]
type dest = [ dest dest aux ]
```

On a ainsi défini les deux types en partageant un maximum de code.

Construire le supertype

```
type all_expr = [
  | all_expr source_aux
  | all_expr dest_aux ]
```

En une ligne, on définit le plus petit supertype de `source` et `dest`.

Construire le supertype

```
type all_expr = [
  | all_expr source_aux
  | all_expr dest_aux ]
```

En une ligne, on définit le plus petit supertype de `source` et `dest`.

```
let rec print_all : all_expr -> unit = function
  | `App (e, e') -> print_all e ; print_all e'
(* [...] Tous les autes cas *)

let coerce_source e = (e : source :> all_expr)
let coerce_dest   e = (e : dest :> all_expr)
```

Des afficheurs incrémentaux

```
let print_common print : _ common -> _ = function
  | `Var v -> print_var v
  | `Uple l -> prints "("; print_list ", " print l; prints ")"

let print_dest_aux print : _ dest_aux -> _ = function
  | #common as e -> print_common print e
  | `App (e, e') -> print e; print e'
  | `Abs (v, e) ->
      prints "fun "; print_var v; prints " -> "; print e

 (* On ferme la récursion, sur les valeurs cette fois *)
let rec print_dest e = print_dest_aux print_dest e
```

En pratique

- Toujours annoter les fonctions

  (donne des avertissements de filtrage, messages d'erreurs lisibles)

- Une coercion est obligatoire si un type fermé doit être unifié avec un type plus grand

- Lire les types inférés avant de passer à la suite (si on n'annote pas)

  On peut très bien écrire une fonction (fausse) qui va recevoir un type trop faible, et qui empêchera le reste du programme de typer

Quand utiliser les variants polymorphes ?

- Quand on ne veut pas déclarer au préalable les constructeurs
- Quand des types inductifs doivent partages des constructeurs
  - Pour des énumérations avec beaucoup de types proches (cf. LablGtk)
  - Avec des types complexes qui se “chevauchent” et pour lesquels l’union a un sens
- Quand une fonction élimine certains des constructeurs
- Comme types fantômes.