Le caractère ` à la rescousse

Factorisation et réutilisation de code grâce aux variants polymorphes

Qui ? Boris Yakobowski

Où ? INRIA Rocquencourt, équipe Gallium

Quand ? 28 Janvier 2008 – JFLA

Plan

Les types inductifs en ML

Les types inductifs :

```
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

Un typage légèrement restreint (1)

- On ne peut pas combiner des types algébriques entre eux

```
type metal_pur = Argent | Plomb | Fer | Or
type alliage_fer = Acier | Fonte

let quantite_fer_pur = function
  | Fer → 1.
  | _ → 0.
let quantite_fer_alliage = function
  | Acier → 0.5
  | Fonte → 0.4
```

Un typage légèrement restreint (1)

- On ne peut pas combiner des types algébriques entre eux

```
type metal_pur = Argent | Plomb | Fer | Or
type alliage_fer = Acier | Fonte

let quantite_fer_pur = function
  | Fer → 1.
  | _ → 0.
let quantite_fer_alliage = function
  | Acier → 0.5
  | Fonte → 0.4
```

```
let quantite_fer : metal_pur + alliage_fer → _ = function
  | (m : metal_pur) → quantite_fer_pur m
  | (m : alliage_fer ) → quantite_fer_alliage m
```

Impossible d'écrire une fonction ayant cette forme

(sans introduire un nouveau type inductif discriminant entre les métaux purs et les alliages)

Un typage légèrement restreint (2)

- On ne peut pas raffiner un type inductif préalablement défini

```
# let transmutation = function
    | Plomb | Fer | Or → Or
    | Argent → Argent

val transmutation : metal_pur → metal_pur = <fun>
```

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

Les variants polymorphes

- Une extension des types inductifs usuels
- Présents dans OCaml
  (depuis la version 2.99)
- De nombreuses extensions, non présentées ici.

  Cf. les travaux de Jacques Garrigue.

Transmutons

```
# let transmutation = function
  | `Plomb | `Fer  | `Or → `Or
  | `Argent → `Argent
```

Les différents métaux “préexistent”

Transmutons

```
# let transmutation = function
  | `Plomb | `Fer | `Or → `Or
  | `Argent → `Argent
```

Les différents métaux “préexistent”

```
val transmutation :
  [< `Argent | `Fer | `Or | `Plomb ] → [> `Argent | `Or ]
```

Le typage n’est pas exact, mais on “sait” que la fonction ne retourne *que* de l’argent ou de l’or.

Transmutons

```
# let transmutation = function
    | `Plomb | `Fer | `Or → `Or
    | `Argent → `Argent
```

Les différents métaux “préexistent”

```
val transmutation :
  [< `Argent | `Fer | `Or | `Plomb ] → [> `Argent | `Or ]
```

Le typage n’est pas exact, mais on “sait” que la fonction ne retourne *que* de l’argent ou de l’or.

```
# let transmutation' = (transmutation :
                [< `Fer | `Or | `Plomb ] → [> `Argent | `Or | `Cuivre ])
```

On peut *affaiblir* le type de `transmutation` par instantiation

Combinons

```
type metal_pur = [ `Argent | `Plomb | `Fer | `Or ]
type alliage_fer = [ `Acier | `Fonte ]
```

On peut utiliser un style très déclaratif ; c'est même conseillé.

Combinons

```
type metal_pur = [ `Argent | `Plomb | `Fer | `Or ]
type alliage_fer = [ `Acier | `Fonte ]
```

On peut utiliser un style très déclaratif ; c'est même conseillé.

Petite parenthèse :

```
# `Argent;;
- : [> `Argent ] = `Argent
# (`Argent : metal_pur );;
- : metal_pur = `Argent
```

`` `Argent `` n'est toujours pas du type `metal_pur`, sauf si on le demande explicitement

Combinons

```
type metal_pur = [ `Argent | `Plomb | `Fer | `Or ]
type alliage_fer = [ `Acier | `Fonte ]
```

On peut utiliser un style très déclaratif ; c'est même conseillé.

```
let quantite_fer_pur : metal_pur → _ = function
  | `Fer → 1.
  | _ → 0.
let quantite_fer_alliage : alliage_fer → _ = [...]
```

Combinons

```
type metal_pur = [ `Argent | `Plomb | `Fer | `Or ]
type alliage_fer = [ `Acier | `Fonte ]
```

On peut utiliser un style très déclaratif ; c'est même conseillé.

```
let quantite_fer_pur : metal_pur → _ = function
  | `Fer → 1.
  | _ → 0.
let quantite_fer_alliage : alliage_fer → _ = [...]
```

```
let quantite_fer : [ metal_pur | alliage_fer ] → _ = function
  | #metal_pur as m → quantite_fer_pur m
  | # alliage_fer as m → quantite_fer_alliage m
```

Bien typé !

Une subtilité : instantiation *vs.* sous-typage

```
# let v = `A;;
val v : [> `A ] = `A
# v = `B;;
- : bool = false
```

La variable de rangée dénotée par le “>” dans `[> `A]` est *implicitement* instanciée, de façon à ce que `v` et `` `B `` aient le type `[> `A | `B]`

Une subtilité : instantiation *vs.* sous-typage

- ```
  # let v = `A;;
  val v : [> `A ] = `A
  # v = `B;;
  - : bool = false
  ```

La variable de rangée dénotée par le “>” dans `[> `A]` est *implicitement* instanciée, de façon à ce que `v` et `` `B `` aient le type `[> `A | `B]`

- ```
  let (a : [`A]) = `A and (b : [`B]) = `B
  ```

Les types de `a` et `b` ne peuvent plus être instanciés, `a = b` est mal-typé

Une subtilité : instantiation *vs.* sous-typage

- ```
  # let v = `A;;
  val v : [> `A ] = `A
  # v = `B;;
  - : bool = false
  ```

La variable de rangée dénotée par le “>” dans `[> `A]` est *implicitement* instanciée, de façon à ce que `v` et `` `B `` aient le type `[> `A | `B]`

- ```
  let (a : [`A]) = `A and (b : [`B]) = `B
  ```

Les types de `a` et `b` ne peuvent plus être instanciés, `a = b` est mal-typé

- ```
  let a' = (a :> [`A | `B]) and b' = (b :> [`A | `B])
  ```

On donne à `a'` et `b'` le type `[ `A | `B ]` *explicitement*, par sous-typage. Le terme `a' = b'` est bien typé.
  ```

Un λ-calcul sans stratégie de réduction

- On part d'un λ-calcul avec let et n-uplets

$$
\begin{array}{rcl}
e & ::= & x \\
 & | & e\ e \\
 & | & \lambda(x)\ e \\
 & | & \text{let } x = e \text{ in } e \\
 & | & (e, ..., e)
\end{array}
$$

Un λ-calcul sans stratégie de réduction

- On part d'un λ-calcul avec let et n-uplets

$$\begin{array}{rcl} e & ::= & x \\ & | & e\ e \\ & | & \lambda(x)\ e \\ & | & \text{let } x = e \text{ in } e \\ & | & (e, ..., e) \end{array}$$

- A priori il est possible d'évaluer les applications et les n-uplets de différentes façons
(par exemple de droite à gauche ou de gauche à droite)

Un λ-calcul sans stratégie de réduction

- On part d'un λ-calcul avec let et n-uplets

$$
\begin{array}{rcl}
e & ::= & x \\
 & | & e\ e \\
 & | & \lambda(x)\ e \\
 & | & \text{let } x = e \text{ in } e \\
 & | & (e, ..., e)
\end{array}
$$

- A priori il est possible d'évaluer les applications et les n-uplets de différentes façons
(par exemple de droite à gauche ou de gauche à droite)

- On peut imposer l'ordre d'évaluation en nommant toutes les sous-expressions avec des let

Un λ-calcul sans stratégie de réduction

- On part d'un λ-calcul avec let et n-uplets
- A priori il est possible d'évaluer les applications et les n-uplets de différentes façons
(par exemple de droite à gauche ou de gauche à droite)
- On peut imposer l'ordre d'évaluation en nommant toutes les sous-expressions avec des let

On obtient un λ-calcul plus simple :

$$
\begin{array}{rcl}
e & ::= & x \\
 & | & x\ x \\
 & | & \lambda(x)\ e \\
 & | & \text{let } x = e \text{ in } e \\
 & | & (x, ..., x)
\end{array}
$$

Avec des variants polymorphes

- Le premier λ-calcul :

```
type expr = [
    | `Var of var
    | `Abs of var * expr
    | `Let of var * expr * expr
    | `App of expr * expr
    | `Uple of expr  list
]
```

- Aucune surprise

Avec des variants polymorphes

- Le λ-calcul avec ordre d'évaluation explicite

```
type expr_n = [
  | `Var of var
  | `Abs of var * expr_n
  | `Let of var * expr_n * expr_n
  | `App of atomic * atomic
  | `Uple of atomic list
]
```

Il faut définir `atomic`

- ```
  type atomic = [ `Var of var ]
  ```

Pas directement `var`
  ```

Traduire d'un langage à l'autre

```
let rec normalize : expr → expr_n = function
  | 'Var _ as v → v
  | 'Abs (v, e) → 'Abs (v, normalize e)
  | 'Let (v, e, e') → 'Let (v, normalize e, normalize e')
  | 'App (e1, e2) →
      let f1, v1 = bind e1 and f2, v2 = bind e2 in
      let r = 'App (v1, v2) in
      f1 (f2 r)
  | 'Uple l → [...]

and bind (* : expr → (expr_n → expr_n) * atomic *) = function
  | 'Var _ as v → (fun e → e), v
  | #expr as e →
      let v' = new_var () in
      (fun e' → 'Let (v', normalize e, e')), ('Var v')
```

Traduire les *n*-uplets

```
| `Uple l →
    (* l' est une liste de couples ``fonction * variable'' *)
    let l' = List.map bind l in
    (* On extrait les variables *)
    let r = `Uple (List.map snd l') in
    (* Et on ajoute les let *)
    List.fold_left (fun e f → f e) r (List.map fst l')
```

On obtient une évaluation de droite à gauche

Traduire les *n*-uplets

```
| `Uple l →
    (* l' est une liste de couples ``fonction * variable'' *)
    let l' = List.map bind l in
    (* On extrait les variables *)
    let r = `Uple (List.map snd l') in
    (* Et on ajoute les let *)
    List.fold_left (fun e f → f e) r (List.map fst l')
```

On obtient une évaluation de droite à gauche

Et de gauche à droite ?

- L'ordre dans lequel les variables fraîches sont liées n'a pas d'importance
- Il suffit de changer l'ordre dans lequel les let sont ajoutés

```
    List.fold_right (fun f e → f e) (List.map fst l') r
```

Les annotations, une aide pour le programmeur

- On a annoté `normalize` explicitement

```
let rec normalize : expr → expr_n = function
```

Le compilateur garantit que les valeurs retournées sont bien normalisées.

Les annotations, une aide pour le programmeur

- On a annoté `normalize` explicitement

```
let rec normalize : expr → expr_n = function
```

Le compilateur garantit que les valeurs retournées sont bien normalisées.

- Le type inféré pour `bind` est “le bon”

```
val bind : expr → (expr_n → expr_n) * atomic *) = <fun>
```

Les annotations, une aide pour le programmeur

- On a annoté `normalize` explicitement

let rec normalize **:** expr $\rightarrow$ expr_n $=$ **function**

Le compilateur garantit que les valeurs retournées sont bien normalisées.

- Le type inféré pour `bind` est "le bon"

val bind **:** expr $\rightarrow$ (expr_n $\rightarrow$ expr_n) $*$ atomic $*$) $=$ <**fun**>

- Pourquoi annoter ?
  - Le compilateur signale les filtrages non exhaustifs ou inutiles, et vérifie que les invariants sont respectés
  - Sans annotation, le type de `normalize` fait une page

Les annotations, une aide pour le programmeur

- On a annoté `normalize` explicitement

let rec normalize **:** expr $\rightarrow$ expr_n $=$ **function**

Le compilateur garantit que les valeurs retournées sont bien normalisées.

- Le type inféré pour `bind` est "le bon"

val bind **:** expr $\rightarrow$ (expr_n $\rightarrow$ expr_n) $*$ atomic $*$) $=$ <**fun**>

- Pourquoi annoter ?
  - Le compilateur signale les filtrages non exhaustifs ou inutiles, et vérifie que les invariants sont respectés
  - Sans annotation, le type de `normalize` fait une page
  - Les annotations sont *toujours facultatives*

Convertir un λ-terme normalisé en un λ-terme général

- L'ensemble des λ-termes dont la stratégie est explicitée est un sous-ensemble des λ-termes génériques

Convertir un λ-terme normalisé en un λ-terme général

- L'ensemble des λ-termes dont la stratégie est explicitée est un sous-ensemble des λ-termes génériques

- La même relation existe sur les types :

```
# let coerce e = (e : expr_n :> expr);;
val coerce : expr_n → expr = <fun>
```

Convertir un λ-terme normalisé en un λ-terme général

- L'ensemble des λ-termes dont la stratégie est explicitée est un sous-ensemble des λ-termes génériques
- La même relation existe sur les types :

```
# let coerce e = (e : expr_n :> expr);;
val coerce : expr_n → expr = <fun>
```

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

À comparer à la traversée récursive qui aurait été nécessaire si on avait utilisé deux types inductifs distincts)

Factorisation de code

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

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

Comparatif

- Quelles constructions sont partagées entres les langages ?
  - Les n-uplets, les variables et les constructions `let` sont les mêmes dans les deux langages
  - Les abstractions et applications sont multiples dans le langage source, et unaires dans le langage cible
  - Le langage cible contient des opérateurs, le langage cible des constantes.

Déclarer les deux types

On factorise *au niveau de la déclaration des types* en paramétrant ces derniers par le type des sous-expressions

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

Déclarer les deux types

On factorise *au niveau de la déclaration des types* en paramétrant ces derniers par le type des sous-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

- Pour obtenir les types `source` et `dest`, il suffit de prendre le plus petit point fixe des types auxiliaires

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

Fermer la récursion

- Pour obtenir les types `source` et `dest`, il suffit de prendre le plus petit point fixe des types auxiliaires

```
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 ]
```

Plus petit sur-langage

- On peut construire très facilement le plus petit supertype contenant les expressions des deux langages

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

Plus petit sur-langage

- On peut construire très facilement le plus petit supertype contenant les expressions des deux langages

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

- Ensuite on définit un afficheur pour ce supertype :

```
let rec print_all : all_expr → unit = function
  | `App (e, e') → print_all e ; print_all e'
  | `SeqApp e l → print_all e; List.iter print_all l
(* [...] Tous les autres cas *)
```

Plus petit sur-langage

- On peut construire très facilement le plus petit supertype contenant les expressions des deux langages

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

- Ensuite on définit un afficheur pour ce supertype :

```
let rec print_all : all_expr → unit = function
  | `App (e, e') → print_all e ; print_all e'
  | `SeqApp e l → print_all e; List.iter print_all l
(* [...] Tous les autres cas *)
```

- Les afficheurs pour les deux langages de départ sont obtenus en injectant ces derniers dans le sur-langage

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

Des afficheurs réutilisables

- On définit plusieurs afficheurs
  - un afficheur par sous-langage
  - chacun prenant en argument l'afficheur pour les sous-expressions

```
let print_common print : 'a common → _ = function
  | `Var v → print_var v
  | `Uple l → prints "(" ; print_list ", " print l ; prints ")"
  | `Let ((v, e), e') → [...]

let print_dest_aux print : 'a 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
```

Des afficheurs réutilisables

- On définit plusieurs afficheurs
  - un afficheur par sous-langage
  - chacun prenant en argument l'afficheur pour les sous-expressions

```
let print_common print : 'a common → _ = function
  | `Var v → print_var v
  | `Uple l → prints "(" ; print_list ", " print l ; prints ")"
  | `Let ((v, e), e') → [...]

let print_dest_aux print : 'a 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, cette fois sur les valeurs *)
let rec print_dest e = print_dest_aux print_dest e
```

En pratique

- Toujours annoter les fonctions

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

- 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

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

Quand utiliser les variants polymorphes ?

- Quand on ne veut pas déclarer au préalable les constructeurs
- Quand des types inductifs doivent partager 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.