Le caractère ‘ à la rescousse – Factorisation et réutilisation de code grâce aux variants polymorphes

Boris Yakobowski

Le caractère ‘ à la rescousse

Factorisation et réutilisation de code grâce aux variants polymorphes

Boris Yakobowski

INRIA Rocquencourt,
http://www.yakobowski.org

Résumé

Les variants polymorphes sont une fonctionnalité puissante du système de types du langage OCaml, dont l'utilisation reste pourtant rare. Dans cet article, nous nous attachons à montrer en quoi ils sont plus expressifs que les types inductifs usuels, à la fois en terme de garantie statique d'invariants et de factorisation de code. Pour cela, nous proposons trois exemples tirés d'un typeur. Aucune connaissance préalable sur les variants polymorphes n'est nécessaire.

1. Introduction

1.1. Le Graal : transformer de façon typée des types inductifs

Une application très courante des langages de la famille ML est de *raffiner* une structure inductive en une autre, plus “précise”, à travers des fonctions de traduction. Malheureusement, lorsque les deux structures sont relativement proches, le programmeur doit souvent choisir entre les garanties statiques offertes par le typage et la verbosité du code, un choix peu enviable.

Appelons en effet τ le type inductif sur lequel on travaille, et f une fonction de transformation. Une première possibilité est de donner à cette fonction le type $\tau \rightarrow \tau$. Cette approche ne reflète toutefois pas (au niveau des types) le fait que certaines constructions de τ peuvent ne plus apparaître dans les valeurs produites par f. En particulier, les fonctions analysant ces valeurs vont typiquement utiliser des assertions **assert false** pour encoder le fait qu'elles “savent” que de telles constructions sont impossibles.

La deuxième solution est de définir un second type inductif τ' encodant les invariants résultant de l'application de f. Malheureusement, avec les types inductifs usuellement disponibles dans les langages ML, certains invariants sont difficilement exprimables. De plus, les fonctions auxiliaires écrites sur τ doivent souvent être entièrement réécrites pour agir sur des valeurs de type τ'.

Une solution pour résoudre (au moins partiellement) ce problème est l'utilisation des variants polymorphes. Ces derniers permettent en effet d'encoder des invariants relativement fins, tout en permettant une très grande factorisation de code.

1.2. Introduction aux variants polymorphes

Les variants polymorphes sont une extension du langage OCaml [8] permettant d'utiliser, sans déclaration préalable, un constructeur. Celui-ci se comporte en surface comme le constructeur d'un type inductif usuel ; en particulier, il peut ou non prendre des arguments. La seule différence syntaxique est la présence d'un caractère apostrophe oblique ‘ devant le nom du constructeur.

```
# 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"
```

Dans cet exemple on vient de définir trois valeurs, a, b1 et b2, dont les valeurs sont respectivement :

1. le constructeur `A ;
2. le constructeur `B avec pour argument[1] un n-uplet de type int * bool ;
3. le constructeur `B avec pour argument la chaîne "foo".

Il est important de remarquer que b1 et b2 "partagent" le constructeur `B, et l'utilisent avec des arguments de types différents. Ceci est rendu possible par l'absence de déclaration des variants polymorphes, ainsi que par leur typage, sur lequel nous allons revenir.

Créons une liste contenant des variants polymorphes.

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

Le type donné à l1 est donc "liste d'éléments de type [> `A | `B of int | `C of bool]". Les crochets autour de ce type, qui sont des délimiteurs syntaxiques (et n'ont donc rien à voir avec le fait qu'on a défini une liste !), peuvent être omis.

Le type [> `A | `B of int | `C of bool ] se lit de la façon suivante : "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" de notre explication, et cache en fait une construction de typage appelée *variable de rangée*. Celle-ci permet d'étendre le type donné à l1.

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

On a ajouté `A et `D aux éléments présents dans l1, et le type de l2 reflète le fait qu'elle contient aussi le constructeur `D (sans argument). Nous n'entrerons pas dans les détails, mais cette extension s'est faite en instanciant la variable de rangée cachée.

Donnons d'autres exemples :

```
# (* Erreur si on ajoute un constructeur avec un type incompatible *)
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
```

La liste n'est pas hétérogène : les constructeurs doivent avoir des arguments compatibles.

```
# (* On décide que le type l1 ne peut plus être étendu *)
let l3 = (l1 : [ `A | `B of int | `C of bool ] list )
val l3 : [ `A | `B of int | `C of bool ] list = [`A; `B 1; `C false]

(* Erreur lorsqu'on essaie d'ajouter un nouveau constructeur *)
let _ = `E :: l3 ;;
            ^^
This expression has type [ `A | `B of int | `C of bool ] list
but is here used with type [> `E ] list
The first variant type does not allow tag(s) `E
```

[1] Contrairement à ceux des inductifs usuels, les constructeurs de variants polymorphes ne sont jamais "aplatis". Ici, `B prend bien en argument le n-uplet (1, true), et pas les deux arguments 1 et true.

Sans variable de rangée, le type de l3 ne peut plus être étendu (au moins par instanciation).

Terminons par la définition d'une fonction :

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

Comme on peut le constater, le filtrage sur les variants polymorphes peut s'écrire de la même façon que sur les inductifs usuels. (On verra néanmoins dans les sections suivantes une nouvelle construction, plus puissante.)

Cet exemple fait apparaître la construction duale de >, qui signifie “au plus” et se note <. La fonction f accepte en argument un type variant contenant au plus les constructeurs `A, `C et `D, et renvoie au moins les constructeurs `B et `D.

On peut instancier l'argument et le type de retour de f pour rendre ce type moins informatif.

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

Étant donné que nos annotations de types ne mentionnent plus de variables de rangées, toute instanciation est devenue impossible. Toutefois, il reste possible d'utiliser l'opérateur de sous-typage :> de OCaml. Néanmoins, son utilisation doit être *explicite*, alors que l'instanciation est implicite.

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

De façon générale, le type [`X | `Y] est un super-type du type [`X ]. En particulier, toute fonction acceptant un argument de type [ `X | `Y ] peut être coercée de façon à recevoir un argument de type [ `X]. En effet, le sous-typage de OCaml est structurel, et utilise la variance des constructeurs, le constructeur flèche étant covariant à droite et contravariant à gauche. Dans notre exemple nous avons réduit le type des arguments acceptés, et étendu le type de retour.

Efficacité des variants La représentation à l'exécution des inductifs usuels et des variants polymorphes a des conséquences sur la compilation des filtrages. Pour les inductifs usuels, des tables de sauts peuvent être utilisées, permettant un branchement en temps constant. A contrario, un filtrage sur n constructeurs variants polymorphes est compilé vers un arbre de sauts, de hauteur $\log(n)$. Par ailleurs, un constructeur de variants polymorphes prenant $k \geq 2$ arguments est toujours[2] représenté comme un constructeur vers un n-uplet de taille k, résultant en une représentation mémoire légèrement moins efficace. Une analyse plus poussée de ces questions se trouve dans [1, § 4].

À propos des exemples Dans la suite, nous utilisons presque systématiquement des annotations de types sur les fonctions que nous définissons, dans la mesure où ce style très déclaratif permet d'obtenir des types plus lisibles, des messages d'erreurs plus clairs, et des avertissements en cas de filtrage non exhaustif. Toutefois, ces annotations sont *toujours* facultatives, l'inférence de types sur les variants polymorphes étant totale.

1.3. Plan

Cet article comporte trois parties, qui introduisent chacunes un exemple (de complexité approximativement croissante). Nous expliquons certaines nouvelles constructions liées aux variants

[2] Cette restriction ne résulte pas d'une impossibilité théorique, mais provient de l'ambiguïté existant dans la syntaxe concrète de OCaml entre un constructeur prenant 2 arguments et un constructeur prenant un n-uplet de 2 éléments.

polymorphes au fur et à mesure de leur introduction ; aussi, et bien que les exemples soient totalement indépendants, une lecture dans l'ordre est conseillée.

La section 2 examine le problème de la résolution des constructeurs de types par un typeur. La section 3 montre comment garantir par typage que certaines valeurs sont en forme normale (pour une relation donnée), à travers l'exemple des types d'un langage avec quantification bornée. La section 4 montre comment "désucrer" de façon typée l'arbre de syntaxe abstraite issue d'une phase d'analyse syntaxique.

Le code source des exemples proposés n'est pas toujours complet, par manque de place. La version intégrale peut être trouvée à l'adresse `http://www.yakobowski.org/jfla08.html`

2. Constructeurs de types non résolus

L'une des tâches d'un typeur est la *résolution de types*. Considérons en effet la déclaration **let** id (x : t) = x. Lorsqu'il rencontre le type t, le typeur doit le résoudre, c'est-à-dire trouver quelle est son identité exacte. En effet, t peut avoir été déclaré dans deux modules ouverts, ou même déclaré deux fois dans le module courant (*e.g.* **type** t = A | B;; **type** t = C | D;;). L'*identité* d'un type doit donc être une information plus précise que son simple nom ; on utilise typiquement un couple comprenant le nom et une valeur unique issue d'un compteur.

Pour fixer les idées, on définit les *constructeurs de types*[3] et les *types* par les grammaires suivantes :

$$C \quad := \quad \texttt{int} \mid \texttt{float} \mid \rightarrow \mid ()^n \mid I^i \qquad\qquad \tau \quad := \quad \alpha \mid C\,\overline{\tau}$$

Le symbole $()^n$ représente le constructeur des n-uplets de longueur n, tandis que I^i représente l'inductif dont le nom est I et l'identité i. Les types de ML sont alors définis inductivement comme étant soit une variable de type α, soit l'application d'un constructeur de types à un ou plusieurs types.

2.1. Des solutions en ML

Les deux grammaires données ci-dessus se traduisent très naturellement en ML :

```
type typeconstr = Int | Float | Arrow | Uple of int | Inductive of inductive_id
type ml_type = Var of var | Constr of typeconstr * ml_type list
```

Nous laissons le type des variables de types non spécifié car il n'a pas d'importance ici. De même, l'identité d'un type inductif n'a pas besoin d'être précisée.

Un inductif pour l'analyse syntaxique Il est difficile, au moment de l'analyse syntaxique, de connaître l'identité des inductifs ; de plus, cela mélangerait typage et analyse syntaxique, une approche qui n'est pas encouragée. La solution traditionnellement retenue est de définir un arbre de syntaxe abstraite spécifique à l'analyse syntaxique, distinct de celui utilisé lors du typage. Le constructeur correspondant aux types inductifs a alors comme paramètre une chaîne de caractères (le nom du type) au lieu d'une identité. On définit également un second type inductif pour les types ML.

```
type parsing_typeconstr = PInt | PFloat | PArrow | PUple of int | PInductive of string
type parsing_ml_type = PVar of var | PConstr of parsing_typeconstr * parsing_ml_type list
```

Étant données ces définitions, une fonction résolvant les constructeurs de types prend en argument un environnement de typage et un objet de type parsing_ml_type, et renvoie un objet de type ml_type (ou lève une exception si un inductif non préalablement défini est rencontré).

[3] D'autres constructeurs de types tels que char, array, ... peuvent bien sûr être ajoutés.

Dans notre algèbre de types simplifiée, la redondance entre les deux définitions est flagrante. Mais elle apparaît également dans des systèmes plus complexes : dans le compilateur OCaml, les types Parsetree.core_type_desc et Types.type_desc ont 10 et 12 constructeurs, et en partagent 7.

Évidemment, cette redondance a un coût. Il faut par exemple écrire un afficheur pour chacun des deux types, avec peu de partage de code possible. De même la fonction résolvant un constructeur de types est composée principalement d'un fastidieux filtrage.

```
let resolve_typeconstr env = function
  | PInt → Int | PFloat → Float | PArrow → Arrow | PUple n → Uple n
  | PInductive s → Inductive (resolve_inductive env s)
```

Ici, les 4 premiers cas sont évidents ; néanmoins, étant données les définitions de typeconstr et parsing_typeconstr, il n'existe pas de solution respectant la discipline de types de ML permettant de les factoriser.

D'un point de vue compilation, le résultat est mitigé. Si les constructeurs des deux types sont donnés *exactement* dans le même ordre, la représentation des inductifs à l'exécution (qui est non typée) permet que les 3 premiers cas de filtrage soient traduits par des opérations *nop*. En revanche, la traductions PUple n → Uple n provoque systématiquement l'allocation d'un nouveau bloc, bien que les deux blocs aient en fait la même représentation mémoire.

Un inductif pour les cas communs Une évolution naturelle est de factoriser dans un troisième type les déclarations communes aux deux inductifs (*i.e.* tous les constructeurs autres que Inductive et PInductive). La fonction resolve_constr devient alors beaucoup moins verbeuse.

```
type common_typeconstr = Int | Float | Arrow | Uple of int
type typeconstr = Common of typeconstr | Inductive of inductive_id
type parsing_typeconstr = PCommon of typeconstr | PInductive of string

let resolve_typeconstr env = function
  | PCommon c → Common c
  | PInductive s → Inductive (resolve_inductive env s)
```

Cette solution est a priori plus séduisante. Toutefois, elle a pour inconvénient d'introduire deux constructeurs (Common et PCommon) qui n'ont sémantiquement aucune raison d'être. Par ailleurs, cette stratification a un coût à l'exécution : tous les constructeurs de types ne correspondant pas aux inductifs doivent subir une indirection. La simplicité d'écriture s'obtient au détriment de l'efficacité et de l'intelligibilité.

Paramétrer typeconstr Une troisième possibilité est de paramétrer typeconstr par le type du constructeur Inductive.

```
type 'a typeconstr_aux = Int | Float | Arrow | Uple of int | Inductive of 'a
type typeconstr = inductive_id typeconstr_aux
type typeconstr_parsing = string typeconstr_aux
```

Cette solution a plusieurs défauts. Tout d'abord, elle ne passe pas vraiment à l'échelle. En effet, elle requiert un paramètre par constructeur prenant des arguments différents lors de l'analyse syntaxique et lors des phases ultérieures. (Dans notre cas, on pourrait par exemple rajouter un constructeur Alias pour les alias de types.) Ensuite, le type typeconstr_aux est finalement “trop” polymorphe : quel serait en effet le sens d'une fonction prenant un argument de type float typeconstr_aux ? Enfin, sans annotation explicite, l'inférence donne souvent aux fonctions des types utilisant typeconstr_aux, peu lisibles. Ici, on veut du polymorphisme *ad hoc* plutôt que du polymorphisme paramétrique.

2.2. Avec des variants polymorphes

Les variants polymorphes permettent de marier harmonieusement les deux premières solutions, en évitant les écueils de la troisième. En effet, ils autorisent le partage des constructeurs communs entre les types typeconstr et typeconstr_parsing.

Nous commençons par mettre en évidence ces constructeurs dans un type common_typeconstr. En dehors des crochets de délimitation et du caractère ‘ précédant les constructeurs, cette déclaration et celle de la solution 2 sont identiques.

```
type common_typeconstr = [ `Int | `Float | `Arrow | `Uple of int ]
```

L'apport d'expressivité apparaît dans la définition de typeconstr et parsing_typeconstr. Il devient en effet possible *d'étendre* la définition ci-dessus, en ajoutant un nouveau constructeur. Notons que l'on a choisi ici d'utiliser le (même) nom `Inductive dans les deux déclarations.

```
type typeconstr = [ common_typeconstr | `Inductive of inductive_id ]
type parsing_typeconstr = [ common_typeconstr | `Inductive of string ]
```

Il est intéressant d'étudier la réponse du typeur OCaml sur ces déclarations.

```
# type typeconstr = [ common_typeconstr | `Inductive of inductive_id ];;
type typeconstr = [ `Arrow | `Float | `Inductive of inductive_id | `Int | `Uple of int ]
# type parsing_typeconstr = [ common_typeconstr | `Inductive of string ];;
type parsing_typeconstr = [ `Arrow | `Float | `Inductive of string | `Int | `Uple of int ]
```

Tout se passe exactement comme si on avait expansé common_typeconstr lors de la définition de typeconstr et parsing_typeconstr, ces deux types semblant “partager” les constructeurs de common_typeconstr. Une telle vision est toutefois trompeuse : comme on l'a vu dans la section 1.2, ces constructeurs “préexistent” et peuvent parfaitement être utilisés en dehors des types qu'on vient de définir. En fait, typeconstr et parsing_typeconstr ne sont rien de plus que des *alias* de types.

Il nous reste à voir comment écrire des fonctions d'affichage, ainsi que resolve_typeconstr. L'écriture d'un afficheur pour common_typeconstr se fait exactement de la même façon qu'avec des inductifs standards (modulo l'ajout des caractères ‘), et de façon très similaire pour typeconstr_parsing.

```
let print_typeconstr_common = function
  | `Int → print_string "int"
  | _ → ...

let print_typeconstr_parsing = function
  | #common_typeconstr as x → print_typeconstr_common x
  | `Inductive s → print_string s
```

La construction #common_typeconstr discrimine en fonction des constructeurs présents dans common_typeconstr : toutes les valeurs commençant par l'un de ces constructeurs (`Int, `Float, `Arrow ou `Uple) sont traitées par la première branche du filtrage. Si au contraire la valeur est de la forme `Inductive (qui n'est pas un constructeur de common_typeconstr), elle est traitée par le deuxième cas. Cet opérateur permet d'effectuer l'opération PCommon c → Common c de la solution 2 de la section précédente, sans marquer les valeurs par les constructeurs Common ou PCommon.

Le type inféré pour la fonction print_typeconstr_parsing est :

```
[< `Arrow | `Float | `Inductive of string | `Int | `Uple of int ] → unit
```

On reconnaît la déclaration expansée de parsing_typeconstr, modulo le caractère <.

Comme le montre la fonction ci-dessous, il est possible (et utile) de contraindre la signature de la fonction lors de sa définition afin d'obtenir des types plus courts (et donc plus lisibles).

```
# let resolve_typeconstr env : parsing_typeconstr → typeconstr = function
  | #common_typeconstr as x → x
  | `Inductive s → `Inductive (resolve_inductive env s)

val resolve_typeconstr : env → parsing_typeconstr → typeconstr
```

Supposons que resolve_typeconstr reçoive comme argument la valeur `Inductive "tree". Si le type tree est dans l'environnement, la fonction va renvoyer la valeur `Inductive id_tree, où id_tree est l'identité du constructeur de types tree. Ainsi, le constructeur `Inductive est utilisé avec deux arguments de types incompatibles, tout cela de manière sûre (et sans qu'il ait été besoin de paramétrer typeconstr par le type de l'argument de `Inductive, comme dans la troisième solution de la Section 2.1).

3. Sous-ensembles garantis statiquement

Le langage ML^F [7] est une extension conservative de ML et du Système F. Dans cette section, nous allons nous intéresser à ses types, et montrer qu'on peut facilement, en utilisant des variants polymorphes, caractériser un ensemble de ces types contenant tous ceux en forme normale. Cet exemple est un peu inhabituel, mais se généralise facilement à de nombreuses fonctions transformant une structure inductive.

3.1. Les types de ML^F

Les types de ML^F sont stratifiés en d'un côté des *monotypes* τ, et de l'autre des *polytypes* σ. Les grammaires suivantes décrivent ces deux classes syntaxiques.

$$\tau \quad ::= \quad \alpha \mid C\,\overline{\tau} \qquad\qquad \sigma \quad ::= \quad \tau \mid \bot \mid \forall\,(\alpha \diamond \sigma)\;\sigma$$

Les monotypes correspondent exactement aux types de ML. Les polytypes sont soit des monotypes, soit le type $\bot$ (qui correspond au type du système F $\forall\alpha.\,\alpha$), soit une forme de quantification bornée $\forall\,(\alpha \diamond \sigma)\;\sigma'$ se lisant très approximativement « σ' dans lequel la variable α parcourt[4] l'ensemble des types décrits par σ » ; α est lié dans σ' mais pas dans σ. On abrège $\forall\,(\alpha \diamond \bot)$ en $\forall\,(\alpha)$.

Cette grammaire se traduit très naturellement, à la fois avec et sans variants polymorphes :

```
type monotype = Var of var | Constr  of typeconstr * monotype list
type polytype = Mono of monotype | Bottom | Bound of (var * polytype) * polytype

type monotype = [ `Var of var  | `Constr of typeconstr * monotype list  ]
type polytype = [ monotype | `Bottom | `Bound of (var * polytype) * polytype ]
```

Toutefois, ces derniers capturent directement l'inclusion entre polytypes et monotypes, et évitent l'indirection introduite par le constructeur Mono. Dans la suite, on ne considérera plus que cette seconde définition.

La recherche des variables libres s'écrit sans aucune surprise[5], modulo l'utilisation de l'opérateur # pour séparer les monotypes des autres constructeurs des polytypes.

```
let rec ftv_monotype : monotype → _ = function
  | `Var v → [v]
  | `Constr (_, l) → List.concat (List.map ftv_monotype l)
```

[4] En réalité, $\diamond$ est une métavariable valant soit $\geq$, soit $=$. Toutefois cette distinction n'a pas d'importance pour l'exemple que nous allons présenter ici.

[5] Notre fonction renvoie des listes afin de simplifier les exemples. Une implémentation réaliste utiliserait des ensembles.

```
and ftv_polytype : polytype → _ = function
  | #monotype as m → ftv_monotype m
  | `Bottom → []
  | `Bound ((v, t), t') → ftv_polytype t @ (list_remove_all v (ftv_polytype t'))
```

Afin d'introduire la suite de notre exemple, nous allons stratifier la définition des polytypes en trois types distincts, les deux premiers encodant le type de $\bot$ et de la quantification bornée. Étant donné que (du point de vue du typage) les définitions de variants polymorphes sont expansées, nos deux définitions de polytype sont parfaitement équivalentes. Nous introduisons également un alias pour les monotypes qui ne sont pas des variables

```
type bot = [ `Bottom ]
type ('a, 'b) bound = [ `Bound of (var * 'a) * 'b ]
type polytype = [ monotype | bot | (polytype, polytype) bound ]

type constr = [ `Constr of typeconstr * monotype list ]
```

3.2. Équivalence entre types de ML^F

ML^F définit sur ses types une relation d'équivalence riche, qui permet notamment d'éliminer les quantifications inutiles. Une version simplifiée des règles les plus importantes est présentée ci-dessous. Les règles sont congruentes, et peuvent donc s'appliquer à l'intérieur d'un type. Étant donné un polytype σ, $\mathsf{ftv}(\sigma)$ représente ses variables libres.

$$\frac{\alpha \notin \mathsf{ftv}(\sigma')}{\forall\,(\alpha \diamond \sigma)\ \sigma' \equiv \sigma'}\ \textsc{Eq-Free} \qquad \textsc{Eq-Var}\quad \forall\,(\alpha \diamond \sigma)\ \alpha \equiv \sigma \qquad \textsc{Eq-Mono}\quad \forall\,(\alpha \diamond \tau)\ \sigma \equiv \sigma[\tau/\alpha]$$

La règle Eq-Free élimine les bornes non utilisées. La règle Eq-Var supprime les quantifications qui ne sont en fait que des alias. Enfin, Eq-Mono exprime le fait que les quantifications sur des monotypes peuvent être expansées ; l'opération de substitution est supposée éviter les captures de variables.

Il est possible de définir une forme normale pour les types. Celle-ci est obtenue en appliquant répétitivement Eq-Free, Eq-Var et Eq-Mono de la gauche vers la droite.

Donnons quelques exemples :

- $\bot$ est une forme normale, par exemple celle du type $\forall\,(\alpha \diamond \sigma)\ \bot$.
- La forme normale du type $\forall\,(\beta \diamond \alpha \to \alpha)\ \forall\,(\gamma \diamond \forall\,(\delta)\ \forall\,(\epsilon)\ \epsilon \to \beta)\ \beta \to \gamma$ est le type $\forall\,(\gamma \diamond \forall\,(\epsilon)\ \epsilon \to (\alpha \to \alpha))\ (\alpha \to \alpha) \to \gamma$. En effet, la variable δ est inutilisée et peut être supprimée par Eq-Free, tandis que la variable β peut être expansée grâce à Eq-Mono.

3.3. Sous-ensembles et sous-types

En étudiant attentivement les règles d'équivalence, on se rend compte que les polytypes en forme normale sont contenus dans le sous-ensemble σ_n de σ décrit par la grammaire suivante :

$$\begin{array}{rcl} \sigma_n & ::= & \tau \mid \bot \mid \forall\,(\alpha \diamond \sigma_\bot)\ \sigma_\tau \\ \sigma_\bot & ::= & \bot \mid \forall\,(\alpha \diamond \sigma_\bot)\ \sigma_\tau \\ \sigma_\tau & ::= & C\,\overline{\tau} \mid \forall\,(\alpha \diamond \sigma_\bot)\ \sigma_\tau \end{array}$$

Il est possible (et aisé) de décrire ces trois ensembles en utilisant les variants polymorphes. A contrario, encoder une telle grammaire avec des inductifs usuels est lourd ; il faut 3 constructeurs pour σ_n, 2 constructeurs pour σ_τ et 2 pour $\sigma_\bot$, ainsi que des fonctions de conversion.

```
type nf_polytype = [ monotype | bot | (bot_poly, constr_poly) bound ]
and bot_poly  =    [            bot | (bot_poly, constr_poly) bound ]
and constr_poly =  [ constr     |     (bot_poly, constr_poly) bound ]
```

(Malheureusement il ne semble pas possible de factoriser (bot_poly, constr_poly) bound dans un type séparé afin d'éviter la légère duplication de code, le typeur de OCaml rejetant une telle déclaration.)

Comme on l'a vu, σ_n est un sous-ensemble de σ. Utiliser des de variants polymorphes permet de transposer cette relation au niveau des types : nf_polytype est un *sous-type* de polytype ! En pratique, cela veut dire que toute valeur de type nf_polytype peut être *coercée* en une valeur de type polytype, comme le montre la fonction de conversion suivante :

```
let nf_to_polytype t = (t : nf_polytype :> polytype)
```

Cette fonction a pour type nf_polytype → polytype, et convertit donc un polytype en forme normale en un polytype générique. *Cette conversion est purement logique* : l'opérateur de sous-typage :> de OCaml n'a aucun coût à l'exécution, et nf_to_polytype est compilée vers la fonction identité ! C'est là un gain crucial par rapport à une approche utilisant les inductifs usuels, qui aurait demandée une traversée et une reconstruction de tout le terme représentant t.

Fonctions de substitutions Nous commençons par écrire les fonctions substituant un monotype dans les monotypes et les polytypes. Dans ce second cas on suppose que les variables libres du monotype sont différentes des variables liées dans le polytype, afin d'éviter toute capture de variable.

```
# let rec expand_mono_in_mono (v, m) = function
  | `Var v' → if v = v' then m else `Var v'
  | `Constr (c, l) → `Constr (c, List.map (expand_mono_in_mono (v, m)) l)
and expand_mono_in_poly (v, m) : polytype → _ = function
  | #monotype as m' → (expand_mono_in_mono (v, m) m' :> polytype)
  | `Bottom → `Bottom
  | `Bound ((v', t), t') → `Bound ((v', expand_mono_in_poly (v, m) t),
                                   if v = v' then t' else expand_mono_in_poly (v, m) t')

val expand_mono_in_mono : var * monotype → monotype → monotype = <fun>
val expand_mono_in_poly : var * monotype → polytype → polytype = <fun>
```

L'annotation de types sur expand_mono_in_poly est facultative. En revanche, la coercion expand_mono_in_mono (v, m) m' :> polytype est essentielle. En effet, expand_mono_in_mono renvoyant un monotype, en l'absence de coercion la première branche du filtrage imposerait aux autres branches de renvoyer également un monotype. Ceci est contradictoire avec le fait que la fonction peut par exemple renvoyer `Bottom.

Normalisation La fonction de normalisation s'écrit très naturellement. Si le type est un monotype ou ⊥, le résultat est immédiat. Sinon, c'est une quantification bornée, et on normalise successivement ses deux sous-parties, en utilisant au vol toute règle applicable. On suppose qu'aucune variable n'est liée deux fois, et que les variables libres et liées sont disjointes.

```
let rec nf : polytype → nf_polytype = function
  | #bot | #monotype as t → t
  | `Bound ((v, t), t') →
      match nf t with
        | #monotype as m → nf (expand_mono_in_poly (v, m) t') (* Eq−Mono *)
        | #bot_poly as t →
```

```
        match nf t' with
          | `Var v' → if v = v' then t (* Eq−Var *) else `Var v' (* Eq−Free *)
          | #bot → `Bottom (* Eq−Free *)
          | #constr_poly as t' →
              let ftv = ftv_polytype (t' : constr_poly :> polytype) in
              if List.mem v ftv then `Bound ((v, t), t')
              else t' (* Eq−Free *)
```

Comme précédemment, l'annotation polytype → nf_polytype est facultative. En revanche, la coercion (t' : constr_poly :>polytype) est nécessaire. En effet, dans ce contexte, t' a pour type constr_poly, qui n'est pas compatible avec polytype (qui est le type des arguments de ftv_polytype). Sans coercion, on obtient le message d'erreur suivant :

```
1  This expression has type
2    [> constr_poly ] =
3      [> `Bound of (var * bot_poly) * constr_poly
4       | `Constr of constr * monotype list
5       | `Var of var ]
6  but is here used with type polytype =
7      [ `Bottom
8      | `Bound of (var * polytype) * polytype
9      | `Constr of constr * monotype list
10     | `Var of var ]
11 Type bot_poly = [ `Bottom | `Bound of (var * bot_poly) * constr_poly ]
12 is not compatible with type polytype =
13     [ `Bottom
14     | `Bound of (var * polytype) * polytype
15     | `Constr of constr * monotype list
16     | `Var of var ]
17 The first variant type does not allow tag(s) `Constr, `Var
```

Détaillons : t' a le type [> constr_poly], qui n'est pas compatible avec le type polytype (lignes 1–10). En effet, les arguments du constructeur `Bound dans les deux types (lignes 3 et 8) sont incompatibles : les types bot_poly et polytype sont incompatibles (lignes 11–16), la raison étant donnée ligne 17.

Sans variants polymorphes Les inductifs traditionnels permettent, au prix d'une certaine lourdeur, d'obtenir le même niveau de garanties statiques que celui obtenu ici. L'auteur a testé cette approche. La fonction nf résultante fait 22 lignes, à comparer avec les 15 de la fonction ci-dessus (soit une augmentation de 46%, sans compter deux fonctions de coercion de 3 lignes chacune) ; elle est également beaucoup moins claire, car elle nécessite l'emploi de deux sous-fonctions auxiliaires.

Par ailleurs, pour obtenir la même complexité à l'exécution, il est impératif d'écrire la fonction ftv_polytype pour qu'elle agisse sur des valeurs de type constr_poly (et donc de l'écrire deux fois). Une solution utilisant une coercion constr_poly ⇒ polytype augmente en effet la complexité d'un facteur $|\tau|$ (où τ est le type coercé), la coercion devant traverser tout le type pour le convertir.

Qu'avons-nous garanti ? Le sous-ensemble σ_n que nous avons caractérisé n'est pas exactement le sous-ensemble des formes normales des types — seulement un sur-ensemble. En effet, l'application de la règle Eq-Free n'est pas capturée par le critère purement syntaxique que nous avons exprimé. Il faudrait probablement pour cela un système avec types dépendants. De façon générale, les variants polymorphes permettent d'encoder facilement tous les invariants expressibles par une grammaire BNF. En revanche, ils n'offrent pas de réelle garantie pour tout ce qui n'est pas expressible à l'aide de ce formalisme, typiquement les questions de lieurs.

4. D'un arbre à l'autre

Dans cette partie, nous allons nous intéresser au "désucrage" d'un arbre de syntaxe abstraite (AST) pour l'analyse syntaxique en un AST correspondant à un langage noyau. Ces deux langages vont partager des constructeurs, et nous allons voir comment écrire un afficheur pour les deux types correspondants.

4.1. Définition des arbres de syntaxe abstraite

Nous partons d'un langage ML usuel, avec plusieurs constructions de haut niveau telles que des opérations arithmétiques, des conditionnelles, des abstractions et applications multiples...

Notre langage cible est un langage noyau comprenant les constructions du lambda-calcul enrichi par les n-uplets, les let, et des constantes. Ce langage est celui utilisé pour le typage (et non pour la compilation, ce qui explique certains de nos choix).

e	::= x	*Variables*	
	| $e\ \overline{e}$	*Application multiples*	
	| $\lambda(\overline{x})\ e$	*Abstractions multiples*	
	| let rec$^?$ $x = e$ in e	*Let (récursif ou non)*	
	| $i \mid b$	*Entiers, Booléens*	
	| $e + e \mid e = e \mid \ldots$	*Opérations arithmétiques*	
	| if e then e else e	*Conditionnelles*	
	| $(e, ..., e)$	*N-Uplets*	

ϵ	::= x	*Variables*	
	| $\epsilon\ \epsilon$	*Application*	
	| $\lambda(x)\ \epsilon$	*Abstraction*	
	| let $x = \epsilon$ in ϵ	*Let non récursif*	
	| c	*Constantes*	
	| $(\epsilon, ..., \epsilon)$	*n-uplets*	

Les deux langages partagent de façon "exacte" deux constructions, les variables et les n-uplets. D'autres constructions sont "partiellement" partagées : les let, qui peuvent être récursifs dans le langage source et qui sont toujours non récursifs dans le langage cible, et les constantes, qui sont plus nombreuses dans le langage cible. Elles incluent en effet les entiers et les booléens, mais aussi les opérateurs arithmétiques tels que plus (de type int → int → int), un encodage des conditionnelles (de type $\forall\alpha.\mathsf{bool} \to \alpha \to \alpha \to \alpha$), ...

4.2. Récursion et variants polymorphes

On commence par définir les types des valeurs et des constructeurs de types, en les stratifiant afin de refléter le plus grand nombre de constantes présentes dans l'AST cible.

```
(* Valeurs de base *)
type ground_ct = [ `Int of int | `Bool of bool ]
(* Toutes les constantes *)
type ct = [ ground_ct | `IfTE | `OpPlus | `OpEq | `OpFix ]

(* Constructeurs de types pour les valeurs de base puis les constantes *)
type ground_typeconstr = [ `TInt | `TBool ]
type typeconstr = [ `TVar of var | ground_typeconstr | `TArrow of typeconstr * typeconstr ]
```

On introduit ensuite des déclarations auxiliaires pour les constructions let, encodant le fait qu'elles peuvent être ou non récursives. Le constructeur let_aux est paramétré par le type des expressions.

```
type nonrec = [ `NonRec ]
type recnonrec = [ `Rec | nonrec ]
type ('rec_flag, 'expr) let_aux = [ `Let of 'rec_flag * (var * 'expr) * 'expr ]
```

Dans cet exemple, la difficulté principale provient du fait que le type des expressions est récursif, ce qui complique la factorisation de code au niveau de la déclaration des types. La solution est de définir plusieurs sous-types ouverts (*i.e.* paramétrés par le type des expressions), comprenant *certains* des constructeurs. Dans un deuxième temps, on ferme la récursion (au niveau des types) en fusionnant les sous-types ainsi définis.

```
(* Constructeurs communs aux deux AST *)
type 'expr common = [ `Var of var | `Uple of 'expr list ]

(* Constructeurs dans un seul des AST, ou exactement partagés *)
type 'expr source_aux = [
  | 'expr common
  | `SeqApp of 'expr * 'expr list
  | `SeqAbs of var list * 'expr
  | `Plus of 'expr * 'expr | `Eq of 'expr * 'expr
  | `If of 'expr * 'expr * 'expr ]
type 'expr dest_aux = [
  | 'expr common
  | `App of 'expr * 'expr
  | `Abs of var * 'expr ]

(* Les deux AST, obtenus en fermant la récursion *)
type source = [ source source_aux
              | (recnonrec, source) let_aux
              | `Ct of ground_ct * ground_typeconstr ]
type dest =  [ dest dest_aux
              | (nonrec, dest) let_aux
              | `Ct of ct * typeconstr ]
```

La fonction du désucrage est récursive; les applications multiples utilisent toutefois une fonction auxiliaire. Les constantes doivent être coercées vers le type des constantes du langage destination. Les opérateurs arithmétiques et la construction if sont traduits vers l'application de l'opérateur approprié à chaque cas. À titre d'exemple, la valeur ct_plus vaut (`OpPlus, `TArrow (`TInt, `TArrow (`TInt, `TInt))). Les let récursifs utilisent l'opérateur fix pour encoder la récursion.

```
let rec desugar : source → dest = function
  | `Var _ as v → v
  | `Uple l → `Uple (List.map desugar l)
  | `SeqApp (e, l) → desugar_seq_app (desugar e) l
  | `SeqAbs ([], e) → desugar e
  | `SeqAbs (v :: q, e) → `Abs (v, desugar (`SeqAbs (q, e)))
  | `Ct (c, t) → `Ct ((c :> ct), (t :> typeconstr))
  | `If (c, e1, e2) → desugar_seq_app (`Ct ct_if) [c;e1;e2]
  | `Plus (e1, e2) → desugar_seq_app (`Ct ct_plus) [e1;e2]
  | `Eq (e1, e2) → desugar_seq_app (`Ct ct_eq) [e1;e2]
  | `Let (`NonRec, (v, e1), e2) → `Let (`NonRec, (v, desugar e1), desugar e2)
  | `Let (`Rec, (v, e1), e2) →
      `Let (`NonRec, (v, `App (`Ct ct_fix, `Abs (v, desugar e1))), desugar e2)
and desugar_seq_app (e : dest) (* : source list → dest *) = function
  | [] → e
  | e' :: q → desugar_seq_app (`App (e, desugar e')) q
```

4.3. Le meilleur des deux arbres

Écrire un afficheur pour chacun des deux AST peut se faire de trois façons, ordonnées ici par factorisation de code croissante :

1. Écrire deux afficheurs distincts, un pour chaque type, sans factoriser de code. Ici, le code pour les n-uplets, les constantes, les let ou les variables est dupliqué.

   C'est la solution que l'on aurait obtenue si des types inductifs usuels avaient été utilisés. En effet, les deux arbres partageant peu de cas (contrairement aux exemples de la Section 2), il est naturel de définir deux types inductifs séparés.

2. Écrire un afficheur ouvert (prenant en argument un autre afficheur) pour les types common, source_aux et dest_aux, puis écrire un afficheur pour source et dest qui ferme la récursion. Le schéma est exactement le même que celui suivi pour la définition des types, comme le montre l'exemple partiel ci-dessous[6].

```
let print_common print : _ common → unit = function
  | ‘Var v → print_var v
  | ‘Uple l → print_string "(" ; print_list ", " print l; print_string ")"

let print_dest_aux print : _ dest_aux → _ = function
  | #common as e → print_common print e
  | ‘Abs (v, e) → print_string "fun "; print_var v; print_string " → ";
                  print e  (* Appel à l'afficheur passé en argument *)
  | ‘App (e, e') → print e ; print e' (* Idem *)

let rec print_dest : dest → _ = function
  | #dest_aux as e → print_dest_aux print_dest e (* Récursion *)
  | #let_aux as l → print_let print_dest l
  | ‘Ct (c, _) → [...]
```

   Cette approche permet d'obtenir du code réellement extensible.

3. La dernière possibilité, la plus adaptée ici, consiste à définir le plus petit supertype des deux AST, et à écrire un afficheur pour ce type particulier. De cette façon, le partage de code est optimal (en particulier aucune fonction auxiliaire n'est nécessaire). Les deux AST étant des sous-types de ce supertype, on obtient “gratuitement” les fonctions de conversion. De plus, grâce à la façon dont nous avons structuré nos types, l'écriture du supertype est immédiate.

```
type all_expr = [  (* Sur-type des deux AST *)
  | all_expr source_aux
  | all_expr dest_aux
  | (recnonrec, all_expr) let_aux
  | ‘Ct of ct * typeconstr ]

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

   On notera au passage que le sous-typage de OCaml se fait en profondeur, comme le montre le constructeur ‘Let qui prend comme premier argument un objet de type [‘NonRec] dans dest et un objet de type [‘Rec | ‘NonRec] dans all_expr.

   L'écriture de l'afficheur est extrêmement naturelle en utilisant cette méthode. En particulier il n'est pas nécessaire d'écrire des afficheurs ouverts, la récursion pouvant être fermée directement.

[6] Nous ne traitons pas ici le problème consistant à reparenthèser les expressions, qui est orthogonal.

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

let print_source e = print_all (coerce_source e)
let print_dest e = print_all (coerce_dest e)
```

Dans le cas général, la solution 2 est toujours applicable et est à notre avis toujours préférable à la solution 1. La solution 3 ne s'applique que si le supertype a toujours un "sens". C'est le cas ici, dans la mesure où l'on peut facilement mélanger les constructions des deux AST. Elle est plus brève, mais moins extensible si d'autres variantes de l'AST sont écrites.

La solution 3 impose toutefois un peu de rigueur lors de la définition des AST intermédiaires. En effet, du fait que `Ct et `Let prennent des arguments différents dans les deux AST, si on ajoute ces constructeurs directement dans les types source_aux et dest_aux, il n'est pas possible d'obtenir le supertype automatiquement en fermant la récursion (il faut réécrire tout le type "à la main"). C'est la raison pour laquelle on les a "sortis" de ces définitions dans la section 4.2.

Conclusion

Les variants polymorphes au quotidien L'auteur emploie très régulièrement des variants polymorphes dans ses développements. D'après son expérience, les difficultés rencontrées sont de trois sortes :

1. La complexité des types inférés (en particulier, ceux-ci sont souvent récursifs).
2. La longueur des messages d'erreurs lorsqu'une expression n'est pas typable.
3. Faut-il mettre une coercion à un endroit précis du code ?

En pratique, l'ajout d'annotations de types sur les fonctions récursives fait quasiment disparaître la première difficulté, et atténue la deuxième ; l'exemple de la section 3.3 montre qu'un message d'erreur sur des abréviations de types est tout à fait intelligible. Pour le troisième point, le programmeur doit avoir à l'esprit les types des fonctions sur lesquelles il travaille, et insérer une coercion dès qu'une fonction reçoit une valeur qui a un type plus contraint que le type attendu. Toutefois, si les fonctions sont annotées, le moteur d'inférence signale généralement une erreur au "bon" endroit.

Travaux connexes Jacques Garrigue [1] présente la sémantique et le schéma de compilation des variants polymorphes dans OCaml ; la section 3 donne des exemples de leur utilisation. Les exemples proposés dans cet article correspondent à la section 3.2 "Polymorphism and subtyping". D'autres exemples, qualifiés de "monomorphes" (section 3.1) sont utilisés par exemple dans la bibliothèque Lablgtk [5]. Dans un autre article [2], Garrigue décrit une solution au problème de l'extension, à travers l'exemple d'un évaluateur de λ-calcul avec constantes. Nous avons utilisé la même approche pour le point 2 de la section 4.3. Dans un troisième article [4], Garrigue décrit le typage du filtrage en présence de variants polymorphes. La spécification formelle du typage des variants polymorphes dans OCaml peut être trouvée dans [3]. Kagawa [6] propose une implémentation des variants polymorphes au-dessus du système de types de Haskell.

Conclusion L'auteur espère que cet article contribuera à démystifier les variants polymorphes, à la fois au niveau de la difficulté de leur utilisation, et de leur intérêt. Nous voudrions insister sur le fait que l'écriture de fonctions telles que nf (section 3.3), desugar (section 4.2) ou print_all (section 4.3) est très naturelle. En pratique, il est même *plus* facile d'écrire certaines fonctions (telles que nf), le système de types garantissant les invariants utilisés ; avec une approche plus traditionnelle le programmeur doit avoir en tête les cas impossibles, et les éliminer manuellement en utilisant des assertions.

Remerciements L'auteur tient à remercier Jacques Garrigue pour ses remarques et explications, Zaynah Dargaye et Benoit Razet pour leurs suggestions et leur relecture attentive, et les relecteurs anonymes pour leurs commentaires et idées.

Références

[1] Jacques Garrigue. Programming with polymorphic variants. In *ML Workshop*, Baltimore, September 1998.

[2] Jacques Garrigue. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering*, Sasaguri, Japan, November 2000.

[3] Jacques Garrigue. Simple type inference for structural polymorphism. In *Ninth International Workshop on Foundations of Object-Oriented Languages*, Portland, Oregon, 2002.

[4] Jacques Garrigue. Typing deep pattern-matching in presence of polymorphic variants. In *JSSST Workshop on Programming and Programming Languages*, Gamagori, Japan, March 2004.

[5] Jacques Garrigue, Benjamin Monate, Olivier Andrieu, Jun Furuse, Maxence Guesdon, and Stefano Zacchiroli. Lablgtk, an Objective Caml interface to Gtk+. Available at `http://wwwfun.kurims.kyoto-u.ac.jp/soft/olabl/lablgtk.html`.

[6] Koji Kagawa. Polymorphic variants in Haskell. In *Haskell '06 : Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, pages 37–47, New York, NY, USA, 2006. ACM Press.

[7] Didier Le Botlan and Didier Rémy. MLF : Raising ML to the power of System-F. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 27–38, August 2003.

[8] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The Objective Caml system release 3.10, Documentation and user's manual, May 2007.