

Programmation Avancée

“Dites le avec des types!”

David Baelde

ENS Paris-Saclay, L3 2020–2021

Deux nouveaux traits

- Variants polymorphes (OCaml 3+, voir aussi CDuce)
↔ polymorphisme et sous-typage structurels
- Types algébriques généralisés (OCaml 4+, ext. Haskell, autres)

Deux nouveaux traits

- Variants polymorphes (OCaml 3+, voir aussi CDuce)
↔ polymorphisme et sous-typage structurels
- Types algébriques généralisés (OCaml 4+, ext. Haskell, autres)

Slogan

Type = Invariant

Deux nouveaux traits

- Variants polymorphes (OCaml 3+, voir aussi CDuce)
↔ polymorphisme et sous-typage structurels
- Types algébriques généralisés (OCaml 4+, ext. Haskell, autres)

Slogan

Type = Invariant

- Spécifier plus fortement le code :
plus lisible, plus sûr car vérifié par le compilateur.
- Plus le typage statique est fort,
plus on peut optimiser le code généré.

Variants polymorphes

Variants standards

Synonymes : types sommes, algébriques ou variants.

```
type 'a list = Nil | Cons of 'a * 'a list
```

En déclarant le type, on déclare de nouvelles valeurs.

La valeur `Nil` n'appartient qu'au type `list`,
et n'existe pas avant sa déclaration.

Variants polymorphes

Les valeurs existent sans déclaration préalable...

```
# 'Nil ;;  
- : [> 'Nil ] = 'Nil  
# 'Foo 2 ;;  
- : [> 'Foo of int ] = 'Foo 2  
# 'Nil "blah" ;;  
- : [> 'Nil of string ] = 'Nil "blah"
```

Variants polymorphes

Les valeurs existent sans déclaration préalable...

```
# 'Nil ;;
- : [> 'Nil ] = 'Nil
# 'Foo 2 ;;
- : [> 'Foo of int ] = 'Foo 2
# 'Nil "blah" ;;
- : [> 'Nil of string ] = 'Nil "blah"
```

...les types caractérisent des ensembles de valeurs.

```
type t = [ 'Cons of int * t | 'Nil ]
let empty : t = 'Nil
let l : t = 'Cons (3, 'Nil)
```


Les valeurs existent sans déclaration préalable...

```
# 'Nil ;;  
- : [> 'Nil ] = 'Nil  
# 'Foo 2 ;;  
- : [> 'Foo of int ] = 'Foo 2  
# 'Nil "blah" ;;  
- : [> 'Nil of string ] = 'Nil "blah"
```

...les types caractérisent des ensembles de valeurs.

```
type t = [ 'Cons of int * t | 'Nil ]  
let empty : t = 'Nil  
let l : t = 'Cons (3, 'Nil)
```

- Syntaxe proche mais typage nettement différent.
- Introduit dans OCaml 3 par Jacques Garrigue, pas d'équivalent immédiat dans d'autres langages.

Sous-typage structurel

On retrouve les mêmes ingrédients qu'avec les objets.

Sous-typage, en apparence

```
type toute_saison = [ 'Patate | 'Laitue ]
type ete = [ toute_saison | 'Cerise | 'Pêche ]
type hiver = [ toute_saison | 'Betterave | 'Courge ]

let panier_ete () : ete list =
  'Laitue ::
  [ if Random.bool () then 'Cerise else 'Pêche ]
```

Sous-typage via coercions explicites

```
let poids : [ete|hiver] list -> int = ...
let f (x : ete list) = poids (x :> [ete|hiver] list)
```

Polymorphisme et rangées

Au moins ceci

```
# let e = 'Nil ;;
val e : [> 'Nil ] = 'Nil
# let g = function 'Toto -> "toto" | _ -> "xxx" ;;
val g : [> 'Toto ] -> string = <fun>
# let f x = if x = 'Toto then 'Tutu else x ;;
val f : ([> 'Toto | 'Tutu ] as 'a) -> 'a = <fun>
```

Au plus cela

```
# let g = function 'A -> "A" | 'B -> "B" ;;
val g : [< 'A | 'B ] -> string = <fun>
```

Polymorphisme et rangées

Au moins ceci

```
# let e = 'Nil ;;
val e : [> 'Nil ] = 'Nil
# let g = function 'Toto -> "toto" | _ -> "xxx" ;;
val g : [> 'Toto ] -> string = <fun>
# let f x = if x = 'Toto then 'Tutu else x ;;
val f : ([> 'Toto | 'Tutu ] as 'a) -> 'a = <fun>
```

Au plus cela

```
# let g = function 'A -> "A" | 'B -> "B" ;;
val g : [< 'A | 'B ] -> string = <fun>
```

Rangée monomorphe

```
# let c = ref 'A ;;
val c : _ [> 'A ] ref = {contents = 'A}
# c := 'B ; c ;;
- : _ [> 'A | 'B ] ref = {contents = 'B}
```

Quels types pour les fonctions suivantes ?

```
let rec len = function
  | 'Nil -> 0
  | 'Cons (_,l) -> 1 + len l

let tl = function 'Cons (_,l) -> l

let take_3 = function
  | 'Cons (x, 'Cons (y, 'Cons (z, _))) -> x + y + String.
    length z
```

Quels types pour les fonctions suivantes ?

```
let rec len = function
  | 'Nil -> 0
  | 'Cons (_,l) -> 1 + len l

let t1 = function 'Cons (_,l) -> l

let take_3 = function
  | 'Cons (x, 'Cons (y, 'Cons (z, _))) -> x + y + String.
    length z
```

Réponses :

- `val len : ([< 'Cons of 'b * 'a | 'Nil] as 'a)-> int`

Quels types pour les fonctions suivantes ?

```
let rec len = function
  | 'Nil -> 0
  | 'Cons (_,l) -> 1 + len l

let t1 = function 'Cons (_,l) -> l

let take_3 = function
  | 'Cons (x, 'Cons (y, 'Cons (z, _))) -> x + y + String.
    length z
```

Réponses :

- `val len : ([< 'Cons of 'b * 'a | 'Nil] as 'a)-> int`
- `val t1 : [< 'Cons of 'a * 'b] -> 'b`

Quels types pour les fonctions suivantes ?

```
let rec len = function
  | 'Nil -> 0
  | 'Cons (_,l) -> 1 + len l

let t1 = function 'Cons (_,l) -> l

let take_3 = function
  | 'Cons (x, 'Cons (y, 'Cons (z, _))) -> x + y + String.
    length z
```

Réponses :

- `val len : ([< 'Cons of 'b * 'a | 'Nil] as 'a)-> int`
- `val t1 : [< 'Cons of 'a * 'b] -> 'b`
- `val take_3 : [< 'Cons of int * [< 'Cons of int *
[< 'Cons of string * 'a]]] -> int`

Diverses formes d'arbres

```
type 'a tree =  
  [ 'Leaf | 'Node of 'a * 'a tree * 'a tree ]  
type 'a comb =  
  [ 'Leaf | 'Node of 'a * [ 'Leaf ] * 'a comb ]  
  
let comb_of_tree : 'a. 'a tree -> 'a comb =  
  let rec aux = function  
    | 'Node (x,l,r) :: more ->  
      'Node (x, 'Leaf, aux (l::r::more))  
    | 'Leaf :: more -> aux more  
    | [] -> 'Leaf  
  in fun x -> aux [x]
```

Limitations

- Les erreurs de type sont (beaucoup) complexifiées !
 - Annoter son code pour s'y retrouver.

Limitations

- Les erreurs de type sont (beaucoup) complexifiées!
→ **Annoter son code** pour s'y retrouver.
- Coercions “simples” limitées, notamment sur la récursion.
On peut avoir à utiliser une coercion “double” :

```
let f (x : 'a comb) : 'a tree =  
  (x : 'a comb :> 'a tree)
```

Limitations

- Les erreurs de type sont (beaucoup) complexifiées !
→ **Annoter son code** pour s'y retrouver.
- Coercions “simples” limitées, notamment sur la récursion.
On peut avoir à utiliser une coercion “double” :

```
let f (x : 'a comb) : 'a tree =  
  (x : 'a comb :> 'a tree)
```

- Les types ne sont pas raffinés dans les branches de `match`,
il faut expliciter, p.ex. au moyen de la notation `#t` :

```
let cast (x : [ 'Alien | t ]) : t = match x with  
  | 'Alien -> raise Panic  
  | #t as y -> y (* x toujours "trop gros" *)
```

Limitations

- Les erreurs de type sont (beaucoup) complexifiées !
→ **Annoter son code** pour s'y retrouver.
- Coercions “simples” limitées, notamment sur la récursion.
On peut avoir à utiliser une coercion “double” :

```
let f (x : 'a comb) : 'a tree =  
  (x : 'a comb :> 'a tree)
```

- Les types ne sont pas raffinés dans les branches de `match`,
il faut expliciter, p.ex. au moyen de la notation `#t` :

```
let cast (x : [ 'Alien | t ]) : t = match x with  
  | 'Alien -> raise Panic  
  | #t as y -> y (* x toujours "trop gros" *)
```

- Un tag ne peut apparaître qu'une fois dans une rangée.
Le type suivant est interdit :

```
type astarb =  
  [ 'Cons of a * astarb | 'Cons of b * 'Nil ]
```

Le typage reste statique :
aucune info de typage n'est nécessaire à l'exécution.

Les tags sont compilés en un hash : léger surcout.
(Risque de collision de hashes détecté statiquement.)

Utilisations courantes

- Prototypage rapide, duck typing :
ne pas écrire les types... dans un premier temps.

```
let run state = match state with
  | 'Idle -> run (if ... then 'Wait else 'Idle)
  | 'Wait -> let s = get () in run ('Process s)
  | 'Process s -> ...
```

Utilisations courantes

- Prototypage rapide, duck typing :
ne pas écrire les types... dans un premier temps.

```
let run state = match state with
  | 'Idle -> run (if ... then 'Wait else 'Idle)
  | 'Wait -> let s = get () in run ('Process s)
  | 'Process s -> ...
```

- Lisibilité : ne pas lésiner sur les noms explicites.

```
val status : unit -> ['On|'Off] (* > bool *)
```


Utilisations courantes

- Prototypage rapide, duck typing :
ne pas écrire les types... dans un premier temps.

```
let run state = match state with
  | 'Idle -> run (if ... then 'Wait else 'Idle)
  | 'Wait -> let s = get () in run ('Process s)
  | 'Process s -> ...
```

- Lisibilité : ne pas lésiner sur les noms explicites.

```
val status : unit -> ['On|'Off] (* > bool *)
```

- Réutilisation de constructeurs :

```
type evt = [ 'Mouse of .. | 'Key of .. ]
type widget_evt = [ evt | 'Visible | 'Destroy ]
```

Utilisations courantes

- Prototypage rapide, duck typing :
ne pas écrire les types... dans un premier temps.

```
let run state = match state with
  | 'Idle -> run (if ... then 'Wait else 'Idle)
  | 'Wait -> let s = get () in run ('Process s)
  | 'Process s -> ...
```

- Lisibilité : ne pas lésiner sur les noms explicites.

```
val status : unit -> ['On|'Off] (* > bool *)
```


- Réutilisation de constructeurs :

```
type evt = [ 'Mouse of .. | 'Key of .. ]
type widget_evt = [ evt | 'Visible | 'Destroy ]
```

Les grosses bibliothèques OCaml (Tcl/Tk, Gtk, Ocsigen...) utilisent souvent les variants polymorphes, et les labels — autre addition de J. Garrigue dans OCaml 3.

Un exemple poussé

Enrichissons une implémentation des arbres rouge et noir afin d'**expliquer les invariants** de la structure de données.

 Chris Okasaki, *Red-black trees in a functional setting*, Journal of Functional Programming, 1999 [PDF]

Invariants :

- A. L'arbre est binaire, chaque noeud est rouge ou noir.
- B. Les feuilles sont noires.
- C. La racine est noire.
- D. Un noeud rouge n'a pas de fils rouge.
- E. Tous les chemins partant d'un noeud contiennent le même nombre de noeuds noirs.
- F. L'arbre est un arbre de tri.

La suite dans le code : `redblack/*.ml`

Types algébriques généralisés

Échauffement : types fantômes

Idée : un paramètre de type calculatoirement inutile pour exprimer des contraintes/invariants supplémentaires

Exemple : prévention d'injections de code



Échauffement : types fantômes

Idée : un paramètre de type calculatoirement inutile pour exprimer des contraintes/invariants supplémentaires

Exemple : prévention d'injections de code

```
type clean
type dirty
type 'a str (* = string *)
val input : string -> dirty str
val sanitize : dirty str -> clean str
val eval : clean str -> result
val length : 'a str -> int
```

Note : cela ne fonctionne pas sans type abstrait !

Arbres rouge et noir

Interdire deux noeuds rouges consécutifs :

```
type red
type black
type ('a,'c) t
val leaf : ('a,black) t
val red :
  'a -> ('a,black) t -> ('a,black) t -> ('a,red) t
val black :
  'a -> ('a,'c1) t -> ('a,'c2) t -> ('a,black) t
```

Arbres rouge et noir

Interdire deux noeuds rouges consécutifs :

```
type red
type black
type ('a,'c) t
val leaf : ('a,black) t
val red :
  'a -> ('a,black) t -> ('a,black) t -> ('a,red) t
val black :
  'a -> ('a,'c1) t -> ('a,'c2) t -> ('a,black) t
```

Critique

- Type abstrait \Rightarrow pas de pattern matching
- Le paramètre de couleur dépend du constructeur
 \Rightarrow inexpressible avec un type variant usuel

Types algébriques

Le mieux qu'on puisse écrire avec un type standard :

```
type ('a,'c) tree =  
  | Leaf  
  | Red of 'a * ('a,black) tree * ('a,black) tree  
  | Black of 'a * ('a,'c) tree * ('a,'c) tree
```

Limitations

- En sortie, le paramètre ne dépend pas du constructeur
- En entrée, une seule variable de type disponible

Types algébriques

Le mieux qu'on puisse écrire avec un type standard :

```
type ('a,'c) tree =  
  | Leaf  
  | Red of 'a * ('a,black) tree * ('a,black) tree  
  | Black of 'a * ('a,'c) tree * ('a,'c) tree
```

Limitations

- En sortie, le paramètre ne dépend pas du constructeur
- En entrée, une seule variable de type disponible

Ce qu'on veut, avec des constructeurs vus comme des fonctions :

```
Leaf : ('a,black) tree  
Red : 'a * ('a,black) tree * ('a,black) tree  
      -> ('a,red) tree  
Black : 'a * ('a,'c) tree * ('a,'c) tree  
        -> ('a,black) tree
```

Types algébriques généralisés

```
type (_,_) tree =  
  | Leaf   : ('a,black) tree  
  | Black  : ('a *  
              ('a,'c1) tree *  
              ('a,'c2) tree) -> ('a,black) tree  
  | Red    : ('a *  
              ('a,black) tree *  
              ('a,black) tree) -> ('a,red) tree
```

Exemples

- `l = Leaf` est un `('a,black)tree`
- `r = Red (1,1,1)` est un `(int,red)tree`
- `Red (2,r,r)` ne type pas car `r` est rouge

Types algébriques généralisés

```
type z
type 'a s
```

```
type (_,_,_) tree =
  | Leaf   : ('a,black,z) tree
  | Black  : ('a *
              ('a,'c1,'h) tree *
              ('a,'c2,'h) tree) -> ('a,black,'h s) tree
  | Red    : ('a *
              ('a,black,'h) tree *
              ('a,black,'h) tree) -> ('a,red,'h) tree
```

Exemples

- $l = \text{Leaf}$ et $r = \text{Red } (1,1,1)$ ont pour hauteur noire z
 - $b = \text{Black } (2,\text{Leaf},r)$ est un $(\text{int},\text{black},z \text{ s})\text{tree}$
 - $\text{Black}(3,r,b)$ ne type pas car r et b n'ont pas même hauteur
 - Aucune contrainte d'ordre n'est exprimée.
- Les mondes du calcul et des types sont encore disjoints.

Pattern matching

La vraie extension se situe au niveau du pattern matching.

Les **types sont raffinés** dans les branches :

```
let rec get_black :  
  type a c h. (a,c,h) tree -> (a,black,h) tree =  
  function  
  | Leaf -> Leaf (* c = black, h = z *)  
  | Black _ as t -> t (* c = black, h = h' s *)  
  | Red (x,a,b) -> get_black a (* c = red *)
```

Note : on utilise une annotation vraiment polymorphe.

Pattern matching

La vraie extension se situe au niveau du pattern matching.

Les **types sont raffinés** dans les branches :

```
let rec get_black :  
  type a c h. (a,c,h) tree -> (a,black,h) tree =  
  function  
  | Leaf -> Leaf (* c = black, h = z *)  
  | Black _ as t -> t (* c = black, h = h' s *)  
  | Red (x,a,b) -> get_black a (* c = red *)
```

Note : on utilise une annotation vraiment polymorphe.

L'**exhaustivité** dépend du type (concision, optim runtime) :

```
let black_get :  
  type a h. (a,black,h s) tree -> a =  
  function Black (x,-,-) -> x (* exhaustif! *)
```

Typage des GADTs

Aucune difficulté pour vérifier les types. Inférer est délicat.

Quel type donner à `f` dans l'exemple suivant ?

```
type _ t = I : int t  
let f = function I -> 3
```

Typage des GADTs

Aucune difficulté pour vérifier les types. Inférer est délicat.

Quel type donner à `f` dans l'exemple suivant ?

```
type _ t = I : int t
let f = function I -> 3
```

On a le choix, mais pas de type principal :

- `int t -> int`
- `'a t -> 'a`
- `'a t -> int`

Contraintes de types

Inférer = résoudre des contraintes de type.

ML standard

- Contraintes sont des équations
- Résolution par unification
- Propriété de mgu \Rightarrow principalité

GADTs

- Contraintes de la forme “equation \Rightarrow equation”
- Exemple précédent : $X = \text{int} \Rightarrow Y = \text{int}$
Deux solutions : $Y = X$ et $Y = \text{int}$
- En général, on peut perdre la décidabilité du typage...
mais pas avec les schémas de types à la ML.

En bref

Vérifier les types est facile, inférer est indécidable.

En pratique

On ne s'en sort pas sans écrire des annotations de typage, mais en général il suffit de spécifier le type des fonctions.

Exemple : printf

Formateurs à un paramètre (version simple, sans continuation) :

```
type _ fmt =  
  | Noop : unit fmt  
  | Int  : 'a fmt -> (int -> 'a) fmt  
  | String : 'a fmt -> (string -> 'a) fmt  
  | Lit  : string * 'a fmt -> 'a fmt  
  
let test : (int -> unit) fmt = Int (Lit ("\n", Noop))
```

Exemple : printf

Formateurs à un paramètre (version simple, sans continuation) :

```
type _ fmt =  
  | Noop : unit fmt  
  | Int : 'a fmt -> (int -> 'a) fmt  
  | String : 'a fmt -> (string -> 'a) fmt  
  | Lit : string * 'a fmt -> 'a fmt  
  
let test : (int -> unit) fmt = Int (Lit ("\n",Noop))  
  
let rec printf : type a. a fmt -> a = function  
  | Noop -> ()  
  | Int fmt -> fun i -> print_int i ; printf fmt  
  | String fmt -> fun s -> print_string s ; printf fmt  
  | Lit (s,fmt) -> print_string s ; printf fmt  
  
let () = printf test 42
```

Exemple : printf

Formateurs à un paramètre (version simple, sans continuation) :

```
type _ fmt =
  | Noop : unit fmt
  | Int : 'a fmt -> (int -> 'a) fmt
  | String : 'a fmt -> (string -> 'a) fmt
  | Lit : string * 'a fmt -> 'a fmt

let test : (int -> unit) fmt = Int (Lit ("\n",Noop))

let rec printf : type a. a fmt -> a = function
  | Noop -> ()
  | Int fmt -> fun i -> print_int i ; printf fmt
  | String fmt -> fun s -> print_string s ; printf fmt
  | Lit (s,fmt) -> print_string s ; printf fmt

let () = printf test 42

let rec scanf : type a. a fmt -> a -> unit = ...
```

Exemple : arbres rouge et noir

Démo : `redblack/redblack4.ml`

Invariants :

- 1 L'arbre est binaire, chaque noeud est rouge ou noir.
- 1 Les feuilles sont noires.
- 2 La racine est noire.
- 2 Un noeud rouge n'a pas de fils rouge.
- 3 Tous les chemins partant d'un noeud contiennent le même nombre de noeuds noirs.
- ∞ L'arbre est un arbre de tri.

Conclusion

Deux nouveaux traits

- Variants polymorphes
- Types algébriques généralisés

Slogan

Exprimez un maximum d'invariants par typage.

C'est plus lisible, sûr et efficace.

- Le manuel de référence, version 4.01, chapitres 4.2 et 7.18
- *Programming with Polymorphic Variants*, Jacques Garrigue, ML Workshop, 1998.
- *Le caractère ‘ à la rescousse*, Boris Yakobowski, JFLA, 2008.
- *Complete and Decidable Type Inference for GADTs*, T. Schrijvers, S. Peyton Jones et M. Sulzmann, ICFP, 2009.