Cours 5 : Typage avancé

2020 - 2021

Thèmes

- types fantômes, types singletons, types uniques, *type-state*
- types (algébriques) non uniformes, généralisés
- types variants polymorphes, extensibles
- types enregistrements, types objets (au sens P.O.O.)

Les types dans OCaml

- le langage est fortement typé
- le système de types est très riche et expressif
- les types peuvent servir de spécification très fine
- on se rapproche des pré/post-conditions
- ces spécifications sont vérifiées par le compilateur
- Conclusion: il faut exploiter les types

le reste du monde

La plupart des constructions présentées existent ou peuvent être reproduites, plus ou moins complètement, dans d'autres langages fortement typés avec polymorphisme paramétrique (par exemple: C++, Java, Rust, F#, Haskell, . . .)

Types fantômes

Définition

Un type fantôme est un type paramétré, dont au moins un des paramètres n'apparaît pas dans la définition des valeurs de ce type

Exemples

- `type 'a t = int`
- `type ('a, 'b) t = Nil | Cons of 'b * ('a, 'b) t`

Usage

- caractériser un état interne/caché (*type-state*)
- sans pénaliser l'exécution (*zero cost abstraction*)

Exemple de type fantôme: spécification

- on veut imposer la lecture du premier caractère d'un fichier
- on définit l'interface FichierLecture1Car
- le paramètre du type _ fichier , prenant les valeurs debut et fin , définit l'état interne du fichier

```
module type FichierLecture1Car =
sig
  type debut
  type fin
  type _ fichier
  val open : string -> debut fichier
  val read : debut fichier -> char * fin fichier
  val close : fin fichier -> unit
end
```

Exemple de type fantôme: réalisation

```
module Impl : FichierLecture1Car =
struct
  type debut = unit
  type fin  = unit
  type _ fichier  = in_channel
  let open nom = open_in nom
  let read f = (input_char f, f)
  let close f = close_in f
end
```

Il est nécessaire d'imposer un usage purement séquentiel du fichier lu

- i.e. interdire:

```
let wrong = let f = Impl.open "toto" in (Impl.read f, Impl.read f, ...)
```

- mais autoriser:

```
let lire_char nom =
let f = Impl.open nom in
let (c, f) = Impl.read f in
Impl.close f;
c;;
```

Types uniques

Définition

Un type unique est un type associé à une seule donnée dans tout le programme. Un type unique est créé en même temps que la valeur correspondante.

Exemples

- avec un type abstrait dans un module:

```
module type TypeUnique =
sig
  type unique
  val obj : unique
end
(* create_unique : 'a -> TypeUnique *)
let create_unique (type a) (v : a) =
  (module struct type unique = a let obj = v end : TypeUnique);;
```

- directement avec des GADT (voir plus loin):

```
type unique = Unique : 'a -> unique;;
let create_unique v = Unique v;;
```

Types uniques

Usage

- caractériser et isoler une donnée unique
- souvent également un type fantôme

Exemple de type unique: problème initial

- on considère un chiffrement asymétrique de type *Rivest-Shamir-Adleman*
- on trouve des interfaces similaires en Java, Scala, etc
- clés publiques/privées de type key de même nature (entiers), mais devraient être distinguées
- la donnée chiffrée de type secret devrait être associée à ses clés

```
module type RSA =
sig
  type key
  type secret
  (* creation d'une paire de cles publique/privee *)
  val create_key_pair : unit -> key * key
  (* chiffrement a l'aide de la cle publique *)
  val encrypt : bytes -> key -> secret
  (* dechiffrement a l'aide de la cle privee *)
  val decrypt : secret -> key -> bytes
end
```

Exemple de type unique: spécification

- un type ('typ, 'uniq) key et un type 'uniq secret
- une interface KeyPair spécifiant les modules avec un type unique + 2 clés
- une fonction create_key_pair renverra un tel module

```
module type RSA =
sig
 type pub
 type priv
  type ('typ, 'uniq) key
  type 'uniq secret
  module type KeyPair =
    sig
      type unique
      val pubk : (pub, unique) key
      val privk : (priv, unique) key
    end
  (* creation d'une paire de cles publique/privee *)
  val create_key_pair : unit -> (module KeyPair)
  (* chiffrement a l'aide de la cle publique *)
  val encrypt : bytes -> (pub, 'uniq) key -> 'uniq secret
  (* dechiffrement a l'aide de la cle privee *)
  val decrypt : 'uniq secret -> (priv, 'uniq) key -> bytes
end
```

Exemple de type unique: réalisation

- une réalisation “stub”, sans aucun chiffrement

```
module NullCrypt : RSA =
  struct
    type pub
    type priv
    type ('typ, 'uniq) key = unit
    type 'uniq secret = bytes
    module type KeyPair =
    sig
      type unique
      val pubk : (pub, unique) key
      val privk : (priv, unique) key
    end
    let create_key_pair () = (module
                                struct
                                  type unique
                                  let pubk = ()
                                  let privk = ()
                                end : KeyPair)
    let encrypt by pk = by;;
    let decrypt se pk = se;;
  end
```

Exemple de type unique: réalisation

- une application classique
- garantie sans erreurs de clé

```
let _ =
  let msg = Bytes.of_string "message super important" in
  let (module Key1) = NullCrypt.create_key_pair () in
  let (module Key2) = NullCrypt.create_key_pair () in
  let msg_secret = NullCrypt.encrypt msg Key1.pubk in
  let msg_decode = NullCrypt.decrypt msg_secret Key1.privk (*erreur si mauvaise cle *) in
  msg_decode = msg;;
```

Types non uniformes

Définition

Un type (récursif) non uniforme `'a t` fait apparaître des instances différentes du paramètre dans sa définition, **fonctions** de `'a`.

Exemples

- listes alternées:

```
type ('a, 'b) alt_list = | Nil | Cons of 'a * ('b, 'a) alt_list ;;
```

- arbres binaires équilibrés:

```
type 'a perfect_tree = | Empty | Node of 'a * ('a * 'a) perfect_tree ;;
```

Usage

- représenter des invariants de structure “descendants”
- meilleure spécification
- associé au polymorphisme explicite

Exemples de type non uniforme: les listes alternées

- to_alt_list : transformer une liste de paires en une liste alternée
- from_alt_list : couper une liste alternée en deux listes
- déclarations de types différentes (récursion uniforme *vs.* non uniforme)

```
type ('a, 'b) alt_list = | Nil | Cons of 'a * ('b, 'a) alt_list ;;

let rec to_alt_list : ('a * 'b) list -> ('a, 'b) alt_list =
  function | [] -> Nil
           | (a, b)::q -> Cons (a, Cons (b, to_alt_list q));;

let rec from_alt_list : 'a 'b. ('a, 'b) alt_list -> 'a list * 'b list =
  function | Nil -> ([], [])
           | Cons (a, q) -> let (qb, qa) = from_alt_list q in (a::qa, qb);;
```

Types algébriques généralisés

Définition

Les types algébriques généralisés [1] permettent de choisir les paramètres de type librement. Syntaxe et inférence sont distincts des autres types.

Exemples

- types uniques (existentiels):

```
type stringable = Stringable : 'a * ('a -> string) -> stringable
```

- interprète bien typé:

```
type _ repr =
| Int : int -> int repr
| Add : (int -> int -> int) repr
let eval : type a. a repr -> a =
  function
  | Int i -> i
  | Add -> (fun a b -> a+b)
```

[1] *Generalized Algebraic Data Type*

Types algébriques généralisés

Usage

- généralisation des types non uniformes
- permet d'exprimer les types uniques
- permet d'exprimer le *Run Time Type Information*
- points communs avec le *template metaprogramming* de C++
- très expressif, associé au mécanisme d'**exhaustivité du filtrage**

Types algébriques généralisés: application

Problème

- définition des projections génériques d'un triplet, par une seule fonction
- point de départ possible, avec proj_triple : triple_proj -> ('a*'a*'a) ->'a:

```
(* les 3 composantes *)
type triple_proj =
| First  : triple_proj
| Second : triple_proj
| Third  : triple_proj
(* les 3 projections primitives *)
let first  (a, b, c) = a
let second (a, b, c) = b
let third  (a, b, c) = c
(* la fonction de projection unique *)
let proj_triple = function
  | First  -> first
  | Second -> second
  | Third  -> third
```

Types algébriques généralisés: application

Introduction des GADT

- `triple_proj` prend un paramètre, correspondant au type de la projection
- la définition de `proj_triple` est polymorphe (**type** proj) et possède un type différent par constructeur/branche de filtrage
- `proj_triple : 'a proj_triple -> 'a`, a maintenant le bon type
- `( proj_triple First ) : 'a * 'b * 'c -> 'a`, a également le bon type

```
type _ triple_proj =
| First  : ('a * 'b * 'c -> 'a) triple_proj
| Second : ('a * 'b * 'c -> 'b) triple_proj
| Third  : ('a * 'b * 'c -> 'c) triple_proj

let proj_triple : type proj. proj triple_proj -> proj = function
    | First  -> first
    | Second -> second
    | Third  -> third
```

Types algébriques généralisés: modélisation de problèmes

- on modélise, par des types, le problème *Homme-Loup-Mouton-Chou*
- *H*, sur une barque à 2 places, doit transporter *L*, *M* et *C*, de la rive gauche à la rive droite
- *M* ne peut rester seul (sans *H*) sur une rive avec *L* ou *C*

```
(* les positions : rive gauche ou rive droite *)
type g = private G;;
type d = private D;;

(* les 4 mouvements possibles des 4 entites h, l, m, c *)
type ('h, 'l, 'm, 'c, 'h1, 'l1, 'm1, 'c1) move =
  | H  : ('h, 'l, 'm, 'c, 'h1, 'l, 'm, 'c) move
  | HL : ('h, 'h, 'm, 'c, 'h1, 'h1, 'm, 'c) move
  | HM : ('h, 'l, 'h, 'c, 'h1, 'l, 'h1, 'c) move
  | HC : ('h, 'l, 'm, 'h, 'h1, 'l, 'm, 'h1) move;;

(* condition de securite : m ne mange pas c et n'est pas mange par l *)
type ('h, 'l, 'm, 'c) safe =
  | SafeHM : ('h, 'l, 'h, 'c) safe
  | SafeHLC : ('h, 'h, 'm, 'h) safe;;
```

Types algébriques généralisés: *SAT-solving*

- un chemin est soit terminé (tous sur la rive droite), soit un premier mouvement sûr suivi du chemin restant
- exists_path permet de tester l'absence de solution, par des clauses de **réfutation**, qui sont vérifiées par le système de types
- le dernier filtre est refusé, il existe bien une solution de longueur 7

```
(* les chemins, listes de mouvements surs *)
type ('h, 'l, 'm, 'c) path =
  | OK : (d, d, d, d) path
  | GD : (g, 'l, 'm, 'c, d, 'l1, 'm1, 'c1) move * (d, 'l1, 'm1, 'c1) safe
         * (d, 'l1, 'm1, 'c1) path -> (g, 'l, 'm, 'c) path
  | DG : (d, 'l, 'm, 'c, g, 'l1, 'm1, 'c1) move * (g, 'l1, 'm1, 'c1) safe
         * (g, 'l1, 'm1, 'c1) path -> (d, 'l, 'm, 'c) path;;

let exists_path (p : (g, g, g, g) path) =
  match p with
  | GD (_, _, OK) -> .
  | GD (_, _, DG (_, _, GD (_, _, OK))) -> .
  | GD (_, _, DG (_, _, GD (_, _, DG (_, _, GD (_, _, OK))))) -> .
  | GD (_, _, DG (_, _, GD (_, _, DG (_, _, GD (_, _, DG (_, _, GD (_, _, OK))))))) -> .
```

Types algébriques généralisés: résolution de problèmes

- un chemin est soit terminé (tous sur la rive droite), soit un premier mouvement sûr suivi du chemin restant
- exists_path permet de tester l'absence de solution, par des clauses de **réfutation**, qui sont vérifiées par le système de types
- le dernier filtre est refusé, il existe bien une solution de longueur 7

```
(* les chemins, listes de mouvements surs *)
type ('h, 'l, 'm, 'c) path =
  | OK : (d, d, d, d) path
  | GD : (g, 'l, 'm, 'c, d, 'l1, 'm1, 'c1) move * (d, 'l1, 'm1, 'c1) safe
         * (d, 'l1, 'm1, 'c1) path -> (g, 'l, 'm, 'c) path
  | DG : (d, 'l, 'm, 'c, g, 'l1, 'm1, 'c1) move * (g, 'l1, 'm1, 'c1) safe
         * (g, 'l1, 'm1, 'c1) path -> (d, 'l, 'm, 'c) path;;

let exists_path (p : (g, g, g, g) path) =
  match p with
  | GD (_, _, OK) -> .
  | GD (_, _, DG (_, _, GD (_, _, OK))) -> .
  | GD (_, _, DG (_, _, GD (_, _, DG (_, _, GD (_, _, OK))))) -> .
  | GD (_, _, DG (_, _, GD (_, _, DG (_, _, GD (_, _, DG (_, _, GD (_, _, OK))))))) -> .
```

Types algébriques extensibles

Définition

Un type algébrique extensible est un type auquel on peut dynamiquement ajouter des contructeurs. Sa définition évolue au cours du temps.

Exemple

```
type t = ..          (* type extensible initialement vide *)
⋮
type t += 
| Cas1 : ... -> t (* ajout de Cas1 au type t *)
⋮
type t += 
| Cas2 : ... -> t (* ajout de Cas2 au type t *)
```

Usage

- mécanisme proche du sous-typage “orienté objet”
- oblige à prévoir un cas de filtrage par défaut

Types variants polymorphes

Définition

Un type variant polymorphe est simplement un constructeur typé, qui forme un type algébrique à lui seul. Les types algébriques usuels sont formés par agrégation de types variants. Syntaxe et inférence sont distincts.

Exemples

- `type 'a vlist = [ `Nil | `Cons of 'a * 'a vlist ]`
- `` `Cons (1, `Nil) : [> `Cons of int * [> `Nil ] ] ``
- `` [`A ; `B] : [> `A | `B ] list ``
- `(function `A x -> x=0 | `B -> true) : [< `A of int | `B ] -> bool`

Usage

- types algébriques “ouverts”, pas de définition récursive
- spécifier bornes sup. et inf. des valeurs manipulées
- fonctions partielles
- structures hétérogènes

Types variants polymorphes: les listes

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

let hd l = match l with `Cons (t, q) -> t;;
let hd' (l : [< `Cons of 'a * 'a vlist ]) : 'a = hd l ;;
let rec half l =
  match l with
  | `Nil | `Cons (_, `Nil) -> `Nil
  | `Cons (_, `Cons (t, q)) -> `Cons (t, half q );;
let half' (l : 'a vlist ) : 'a vlist = half l ;;
```

- les types inférés sont parfois complexes
- il est préférable de déclarer les types
- `hd : [< `Cons of 'a * 'b] -> 'a`
- `hd' : [< `Cons of 'a * 'a vlist & 'a * 'b] -> 'a`
- `half : ([< `Cons of 'b * [< `Cons of 'c * 'a | `Nil] | `Nil ] as 'a) -> ([> `Cons of 'c * 'd | `Nil ] as 'd)`
- `half' : 'a vlist -> 'a vlist`

Types enregistrements

- type enregistrement “classique”
- champs mutables ou non
- mise-à-jour impérative (move_x : int -> t -> unit)
- mise-à-jour fonctionnelle (move_y : int -> t -> t)

```
type t = { mutable x : int ; y : int };;

let create x y = { x = x; y = y };;
let move_x d v =
  v.x <- v.x + d;;
let move_y d v =
  { v with y = v.y + d };;
```

Types enregistrements *inlinés*

- type enregistrement “intégré” dans un type algébrique
- constructeurs avec arguments nommés plutôt que tuples anonymes

```
type t =
  | Point of {width: int; mutable x: float; mutable y: float}
  | ...

let v = Point {width = 10; x = 0.; y = 0.}

let scale l = function
  | Point p -> Point {p with x = l *. p.x; y = l *. p.y}
  | ....
let print = function
  | Point {x; y; _} -> Format.printf "%f/%f" x y
  | ....
let reset = function
  | Point p -> p.x <- 0.; p.y <- 0.
  | ...
let invalid = function
  | Point p -> p (* incorrect *)
  | ...
```

Types enregistrements polymorphes

- un type enregistrement peut être paramétrique, par exemple:

```
type ('a, 'b) t = {x: 'a; y: 'b list -> 'b}
let test1 (r : ('a, 'b) t) (e : 'b) : bool = r.y [e] = e
```

- un type enregistrement peut contenir des champs polymorphes
- le champ y est maintenant polymorphe en 'b:

```
type 'a t = {x: 'a; y: 'b. 'b list -> 'b}
let test2 (r : ('a, 'b) t) : bool = (r.y [0] = 0) && (r.y [true] = true)
```

Types objets et classes

- sous-langage objet “classique”
- typage fort, *erasure* $\Rightarrow$ absence du transtypage et de `instanceof`
- membres privés/publiques, méthodes virtuelles, self/super, héritage multiple, interfaces, etc
- typage structurel (non nominal) basé sur les interfaces (avec inférence)

```
class point =
  object
    val mutable x = 0
    method get_x = x
    method move d = x <- x + d
    method print = Format.printf "x = %d" x
  end;;

class colored_point (c : string) =
  object (self)
    inherit point as super
    val c = c
    method color = c
    method print = Format.printf "(%t, c=%s)" (fun _ -> super#print) self#color
  end;;
```

Types objets et classes

- equal_x compare toute paire d'objets de classes quelconques, mais respectant l'interface <get_x : 'a; ... >
- equal_x_point est spécialisée aux objets de la classe point
- equal_p1_x compare tout objet respectant l'interface < get_x : int ; ... > avec le point coloré p1

```
let p0 = new point;;
let p1 = new colored_point "black";;

let equal_x a b = a#get_x = b#get_x;;
let equal_x_point (a : point) (b : point) = equal_x a b;;
let equal_p1_x = equal_x p1;;
```

Conclusion

- OCaml possède un langage de types très puissant
- nombreuses modélisations possibles, purement avec des types
- abstractions à faible coût
- beaucoup d'autres aspects non évoqués