Cours 5 : Typage avancé

2023-2024

Thèmes

- types fantômes, types singletons, types uniques, *type-state*
- types (algébriques) non uniformes, généralisés
- 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 abstraits

Types abstraits

"Type abstrait" : ensemble de valeurs abstraites caractérisé par des opérations et leurs spécifications [Liskov & Zilles, 1974]

- Séparation entre usage et implémentation
- Importance de la notion d'invariant

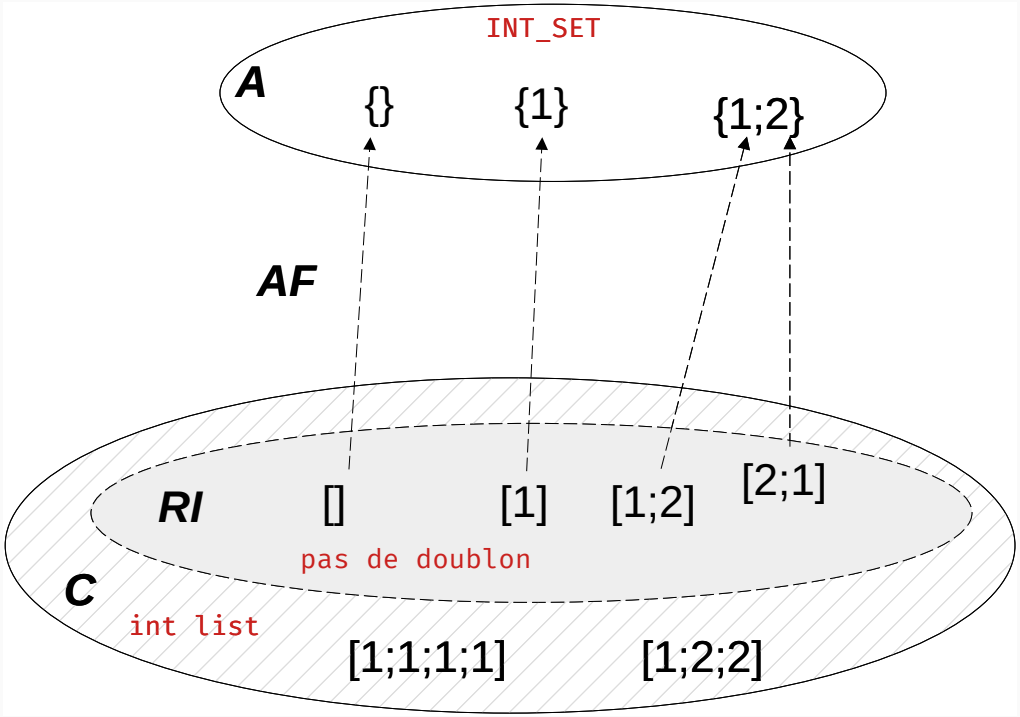
```
module type INT_SET = sig
  type t

  val empty : t
  val member : int -> t -> bool
  val add : int -> t -> t
end
```

Ex : représentation par des listes sans doublon.

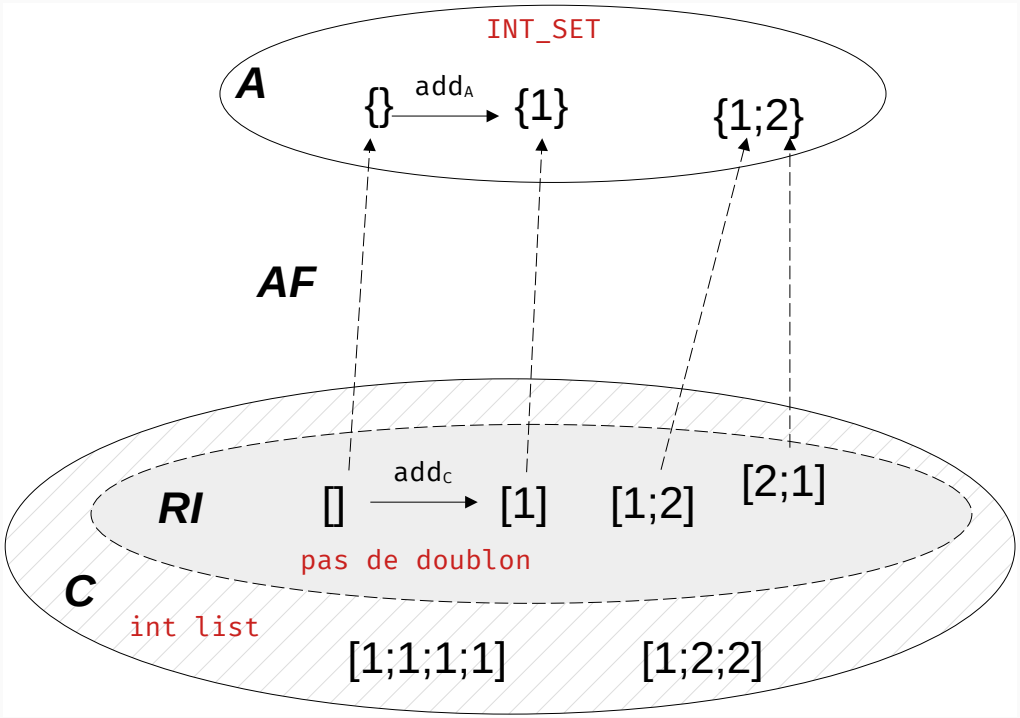
Représentation par un type concret

1. Le type abstrait (A) se situe dans le monde mathématique.
2. Choix d'un type concret C et d'une fonction d'abstraction $AF : C \rightarrow A$
   - AF pas implémentable mais à documenter
   - Surjective
   - Pas forcément injective (cf. $\{1; 2\}$)
   - Généralement partielle, de domaine de définition RI



Représentation par un type concret

1. Le type abstrait (A) se situe dans le monde mathématique.
2. Choix d'un type concret C et d'une fonction d'abstraction $AF : C \to A$
   - AF pas implémentable mais à documenter
   - Surjective
   - Pas forcément injective (cf. $\{1; 2\}$)
   - Généralement partielle, de domaine de définition RI



3. Une opération concrète est correcte si, quand elle respecte ses éventuelles préconditions, elle commute avec AF sur RI.

Invariant de représentation

RI = *invariant* de représentation $\rightsquigarrow$ souvent implémentable.

```
module NoDupList : INT_SET = struct
  type t = int list
  (* AF: la liste [a1; ...; an] représente l'ensemble {a1, ..., an}. *)

  (* En phase de débogage : *)
  let check_rep l = (* RI: la liste ne contient pas de doublons.*)
    if List.(length (sort_uniq Int.compare l) = length l) then l
    else failwith "RI"
  (* En phase d'exploitation : let check_rep l = l
       ou, mieux,  emploi d'une version simplifiée et peu coûteuse *)

  let empty = check_rep []
  let member x s = List.mem x (check_rep s)
  let add x s = check_rep (if member x (check_rep s) then s else x :: s)
end
```

- Faut-il vérifier *RI* pour les arguments des fonctions *d'observation* (cf. deux `check_rep s` ci-dessus) ?
- Quid des fonctions privées ?

Types fantômes

Types fantômes

Définition

Un type fantôme est un type paramétré :

1. dont au moins un des paramètres n'apparaît pas dans la définition des valeurs de ce type
2. dont la définition est abstraite par une signature

Exemples

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

Usage

- caractériser un état interne/caché (*type-state*)
  $\rightsquigarrow$ plutôt pour du code impératif (ou monade d'état, cf. cours 6)
- ne pénalise pas 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
  type fin
  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 non uniformes

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
- nécessite la “récursion polymorphe”

Un type non-uniforme: les arbres parfaits

Fonction qui calcule la profondeur d'un arbre parfait :

```
let rec depth = function
  | Empty -> 0
  | Node (_, sub) -> 1 + depth sub;;
```

Un type non-uniforme: les arbres parfaits

Fonction qui calcule la profondeur d'un arbre parfait :

```
let rec depth = function
  | Empty -> 0
  | Node (_, sub) -> 1 + depth sub;;
Error: This expression has type ('a * 'a) perfect_tree
but an expression was expected of type 'a perfect_tree
The type variable 'a occurs inside 'a * 'a
```

Un type non-uniforme: les arbres parfaits

Fonction qui calcule la profondeur d'un arbre parfait :

```
let rec depth : 'a perfect_tree -> int = function
  | Empty -> 0
  | Node (_, sub) -> 1 + depth sub;;
Error: This expression has type ('a * 'a) perfect_tree
but an expression was expected of type 'a perfect_tree
The type variable 'a occurs inside 'a * 'a
```

Un type non-uniforme: les arbres parfaits

Fonction qui calcule la profondeur d'un arbre parfait :

```
let rec depth : 'a . 'a perfect_tree -> int= function
  | Empty -> 0
  | Node (_, sub) -> 1 + depth sub;;
val depth : 'a perfect_tree -> int = <fun>
```

Récursion polymorphe : chaque application de `depth` doit avoir un type universellement quantifié.

Un type non-uniforme: les arbres parfaits

Fonction qui calcule la profondeur d'un arbre parfait :

```
let rec depth : 'a . 'a perfect_tree -> int= function
  | Empty -> 0
  | Node (_, sub) -> 1 + depth sub;;
val depth : 'a perfect_tree -> int = <fun>
```

Récursion polymorphe : chaque application de `depth` doit avoir un type universellement quantifié.

Variables de type en OCaml : attention aux confusions !

Sous `val` (ex : `val depth : 'a perfect_tree -> int`) une variable de type est forcément quantifiée universellement.

Sous `let`, une variable de type n'est *pas forcément* quantifiée universellement. Elle l'est si :

- Le typeur infère que c'est possible;
- Ou si on l'impose au typeur (ex : `'a .`) et que le typeur parvient à typer la fonction ainsi.

Types algébriques *généralisés*

Changement de notation !

```
type 'a tree =
  | Empty
  | Node of 'a tree * 'a * 'a tree
```

Intuitivement les *constructeurs* `Empty` et `Node` sont des fonctions de profil :

```
Empty : 'a tree
Node : 'a tree * 'a * 'a tree -> 'a tree
```

OCaml offre une notation alternative pour les types algébriques qui reprend cette vision et explicite le type de retour de chaque constructeur :

```
type 'a tree =
  | Empty : 'a tree
  | Node : 'a tree * 'a * 'a tree -> 'a tree
```

Important : jusqu'ici, rien ne change dans le fond p.r. à OCaml "de base"

Changement de notation !

```
type 'a tree =
  | Empty
  | Node of 'a tree * 'a * 'a tree
```

Intuitivement les *constructeurs* `Empty` et `Node` sont des fonctions de profil :

```
Empty : 'a tree
Node : 'a tree * 'a * 'a tree -> 'a tree
```

OCaml offre une notation alternative pour les types algébriques qui reprend cette vision et explicite le type de retour de chaque constructeur :

```
type _ tree =
  | Empty : 'a tree
  | Node : 'a tree * 'a * 'a tree -> 'a tree
```

Important : jusqu'ici, rien ne change dans le fond p.r. à OCaml "de base"

Changement de notation !

```
type 'a tree =
  | Empty
  | Node of 'a tree * 'a * 'a tree
```

Intuitivement les *constructeurs* `Empty` et `Node` sont des fonctions de profil :

```
Empty : 'a tree
Node : 'a tree * 'a * 'a tree -> 'a tree
```

OCaml offre une notation alternative pour les types algébriques qui reprend cette vision et explicite le type de retour de chaque constructeur :

```
type _ tree =
  | Empty : 'b tree
  | Node : 'a tree * 'a * 'a tree -> 'a tree
```

Important : jusqu'ici, rien ne change dans le fond p.r. à OCaml "de base"

Changement de notation !

```
type 'a tree =
  | Empty
  | Node of 'a tree * 'a * 'a tree
```

Intuitivement les *constructeurs* `Empty` et `Node` sont des fonctions de profil :

```
Empty : 'a tree
Node : 'a tree * 'a * 'a tree -> 'a tree
```

OCaml offre une notation alternative pour les types algébriques qui reprend cette vision et explicite le type de retour de chaque constructeur :

```
type _ tree =
  | Empty : _ tree
  | Node : 'a tree * 'a * 'a tree -> 'a tree
```

Important : jusqu'ici, rien ne change dans le fond p.r. à OCaml "de base"

Types algébriques généralisés (GADT)

Définition

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

Deux nouvelles possibilités

- faire varier le type de retour des constructeurs:

```
type _ repr =
  | Int : int -> int repr
  | Add : (int -> int -> int) repr
```

- variables de type n'apparaissant pas dans le type de retour:

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

[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*
- très expressif, associé au mécanisme d'**exhaustivité du filtrage**

Variation du type de retour : application

```
(* types "singletons" pour les entiers de Peano (pas des GADT) *)
type z = Z
type 'n s = S of 'n

(* arbres parfait sous forme GADT *)
type ('a, _) gtree =
  | EmptyG : ('a, z) gtree
  | TreeG : ('a, 'n) gtree * 'a * ('a, 'n) gtree -> ('a, 'n s) gtree
```

Variation du type de retour : application

```
(* types "singletons" pour les entiers de Peano (pas des GADT) *)
type z = Z
type 'n s = S of 'n

(* arbres parfait sous forme GADT *)
type ('a, _) gtree =
  | EmptyG : ('a, z) gtree
  | TreeG : ('a, 'n) gtree * 'a * ('a, 'n) gtree -> ('a, 'n s) gtree

let rec depthG : 'n . (_, 'n) gtree -> 'n = function
| EmptyG -> Z
| TreeG (l, _, _) -> S (depthG l)
```

Variation du type de retour : application

```
(* types "singletons" pour les entiers de Peano (pas des GADT) *)
type z = Z
type 'n s = S of 'n

(* arbres parfait sous forme GADT *)
type ('a, _) gtree =
  | EmptyG : ('a, z) gtree
  | TreeG : ('a, 'n) gtree * 'a * ('a, 'n) gtree -> ('a, 'n s) gtree

let rec depthG : 'n . (_, 'n) gtree -> 'n = function
| EmptyG -> Z
| TreeG (l, _, _) -> S (depthG l)

Error: This pattern matches values of type ('a, $0 s) gtree
but a pattern was expected which matches values of type ('a, z) gtree
```

En OCaml “classique”, les motifs doivent tous avoir le même type ! Ici :

- `EmptyG : ('a, z) gtree`
- `Tree (l, _, _) : ('a, m s) gtree` *pour un certain* `m` (= `$0`)

Variation du type de retour : application

```
(* types "singletons" pour les entiers de Peano (pas des GADT) *)
type z = Z
type 'n s = S of 'n

(* arbres parfait sous forme GADT *)
type ('a, _) gtree =
  | EmptyG : ('a, z) gtree
  | TreeG : ('a, 'n) gtree * 'a * ('a, 'n) gtree -> ('a, 'n s) gtree

let rec depthG : type n . (_, n) gtree -> n = function
| EmptyG -> Z
| TreeG (l, _, _) -> S (depthG l)
val depthG : ('a, 'n) gtree -> 'n = <fun>
```

Nouvelle notation (!) : récursion polymorphe + ajout d'équations de type

- Branche `EmptyG` : ajoute `n` $\equiv$ `z`,
  alors : `Z : z` $\equiv$ `n`
- Branche `TreeG` : ajoute `n` $\equiv$ `m s` (pour un certain `m`),
  alors : `l : (_, m) gtree`, `depthG l : m` et `S (depthG l) : m s` $\equiv$ `n`

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

- directement avec des GADT :

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

- avec un type abstrait et un module de première classe (cf. annexe) :

```
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)
```

Types uniques : exemple

- on considère un chiffrement asymétrique de type *Rivest-Shamir-Adleman*
- 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

Types uniques : déclarations

Interface “classique” :

```
module type RSA = sig


  (* clé *)
  type key
  (* secret produit *)
  type secret
  (* couple de clés publique/privée *)
  type key_pair = Kp of key * key
  (* creation d'une paire de cles publique/privee *)
  val create_key_pair : unit -> key_pair
  (* 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
```

Types uniques : déclarations

Type fantôme :

```
module type RSA = sig
  (* paramètres fantômes pour les clés *)
  type pub
  type priv
  (* clé publique ou privée *)
  type 'pub_or_priv key
  (* secret produit *)
  type secret
  (* couple de clés publique/privée *)
  type key_pair = Kp of pub key * priv key
  (* creation d'une paire de cles publique/privee *)
  val create_key_pair : unit -> key_pair
  (* chiffrement a l'aide de la cle publique *)
  val encrypt : bytes -> pub key -> secret
  (* dechiffrement a l'aide de la cle privee *)
  val decrypt : secret -> priv key -> bytes
end
```

Types uniques : déclarations

Type fantôme + type unique (existentiel) :

```
module type RSA = sig
  (* paramètres fantômes pour les clés *)
  type pub
  type priv
  (* clé publique ou privée avec marqueur 'u pour l'unicité de cette clé *)
  type ('pub_or_priv, 'u) key
  (* secret produit avec la clé de marqueur 'u *)
  type 'u secret
  (* couple de clés publique/privée avec marqueur d'unicité 'u *)
  type key_pair = Kp : (pub, 'u) key * (priv, 'u) key -> key_pair
  (* creation d'une paire de cles publique/privee *)
  val create_key_pair : unit -> key_pair
  (* chiffrement a l'aide de la cle publique *)
  val encrypt : string -> (pub, 'u) key -> 'u secret
  (* dechiffrement a l'aide de la cle privee *)
  val decrypt : 'u secret -> (priv, 'u) key -> string
end
```

Type existentiel

Type de `Kp` : $\forall$ `'u . (pub, 'u) key * (priv, 'u) key -> key_pair`

$\cong$ ($\exists$ `'u . (pub, 'u) key * (priv, 'u) key) -> key_pair`

Quand `Kp` est appliqué, `'u` est connu mais il est abstrait (oublié) dans `key_pair`.

On sait juste qu'il existe une instanciation pour `'u` mais on ne peut la laisser s'échapper sans briser la barrière d'abstraction :

```
module M(R :RSA) = struct
  open R
  let x = let Kp (a,b) = create_key_pair () in 1
end;;
module M : functor (R : RSA) -> sig val x : int end

module M(R :RSA) = struct
  open R
  let x = let Kp (a,b) = create_key_pair () in a
end;;
Error: This expression has type (pub, $Kp_'u) key
       but an expression was expected of type 'a
       The type constructor $Kp_'u would escape its scope
```

Types uniques : implémentation “bidon”

```
module Bidon : RSA = struct
  type pub
  type priv
  type ('pub_or_priv, 'u) key = unit (* int dans une vraie implémentation *)
  type 'u secret = string
  type key_pair = Kp : (pub, 'u) key * (priv, 'u) key -> key_pair
  let create_key_pair () = Kp ((), ())
  let encrypt s pubk = s
  let decrypt secret privk = secret
end
```

Types uniques : implémentation “bidon”

```
let () =
  let open Bidon in
  let msg = "message super important" in
  let Kp (pubk1, privk1) = create_key_pair () in
  let Kp (pubk2, privk2) = create_key_pair () in
  let secret = encrypt msg pubk1 in
  let clair = decrypt secret privk2 in (* erreur *)
  print_endline clair;;
Error: This expression has type (priv, $Kp_'u1) key
       but an expression was expected of type (priv, $Kp_'u) key
       Type $Kp_'u1 is not compatible with type $Kp_'u
```

Types uniques : implémentation “bidon”

```
let () =
  let open Bidon in
  let msg = "message super important" in
  let Kp (pubk1, privk1) = create_key_pair () in
  let Kp (pubk2, privk2) = create_key_pair () in
  let secret = encrypt msg pubk1 in
  let clair = decrypt secret privk1 in
  print_endline clair;;
message super important
```

Enregistrements et objets

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 get_y p = p.y
(* filtrage : *)
let get_y { y; _ } = y

(* mise à jour impérative *)
let move_x d v = v.x <- v.x + d

(* mise-à-jour fonctionnelle *)
let move_y d v = { x = v.x; y = v.y + d }
(* ou mieux : *)
let move_y d v = { v with y = v.y + d }
let move_y d { x; y } = { x; y = 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; f: 'b list -> 'b};;
type ('a, 'b) t = { x : 'a; f : 'b list -> 'b; }
> let test1 r e = r.f [e] = e;;
val test1 : ('a, 'b) t -> 'b -> bool = <fun>
```

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

```
> type 'a u = {x: 'a; f: 'b. 'b list -> 'b};;
type 'a u = { x : 'a; f : 'b. 'b list -> 'b; }
> let test2 r = (r.f [0] = 0) && (r.f [true] = true);;
val test2 : 'a u -> bool = <fun>
```

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 p2 = equal_x p1 p2;;
val equal_x : < get_x : 'a; .. > -> < get_x : 'a; .. > -> bool = <fun>
val equal_x_point : point -> point -> bool = <fun>
val equal_p1_x : < get_x : int; .. > -> bool = <fun>

> let test1 = equal_x p0 p1;;
val test : bool = true
> let test2 = equal_x_point p0 p1;;
Error: This expression has type colored_point
       but an expression was expected of type point
       The second object type has no method color
> let test2 = equal_x_point p0 (p1 :> point);;
val test : bool = true
```

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
  - Variants extensibles
  - Variants polymorphes
  - Types étiquetés et optionnels

Annexe

Exemple de type unique avec un module de première classe

Une interface `KeyPair` spécifie les modules avec un type `unique` + 2 clés.

```
module type RSA = sig
 type pub
 type priv
  type ('typ, 'uniq) key
  type 'uniq secret
  (* Tout module qui implémente cette signature a un type existentiel car,
     étant abstrait, on sait juste qu'un type concret existe qui réalise
     "unique", mais on ne peut savoir lequel. *)
  module type KEY_PAIR = 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 KEY_PAIR)
  (* 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 “bidon”

```
module NullCrypt : RSA = struct
    type pub
    type priv
    type ('typ, 'uniq) key = unit
    type 'uniq secret = bytes
    module type KEY_PAIR = 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 : KEY_PAIR)
    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 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))))))) -> .
```