
TD5: Typage avancé

Thèmes et objectifs

- Types fantômes, types singletons, *type-state*, types non uniformes, GADT, RTTI

▷ Support étudiant

1 Typage avancé

Dans cette section, on s'intéresse à l'expression de propriétés des valeurs d'un langage (ici OCAML) dans le système de types, afin que le compilateur/interprète puisse décharger la preuve à notre place et puisse également mettre en place des optimisations liées à la connaissance de ces propriétés. Tout ce qui suit est implantable, au moins en partie, dans les langages *mainstream* qui supportent la généricité paramétrique, par exemple C++, Java, Rust, ADA, etc.

1.1 Types fantômes

Les types fantômes sont des types paramétrés dont un paramètre n'apparaît pas dans la définition du type. Ce paramètre est donc inutile au sens où il ne représente aucune donnée associée au type défini. Par contre, il peut représenter d'autres informations utiles, comme l'état "interne" d'une valeur de ce type. Le paramètre est donc vu comme une annotation gérée par le compilateur/interprète, à la différence d'un commentaire. On peut notamment décrire des machines à états, des protocoles, etc.

1.1.1 Application: état interne d'un fichier

Dans l'exemple suivant, on décrit le type des fichiers dans lesquels on doit lire un unique caractère, au moyen d'un automate à 2 états `debut` et `fin`.

```
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
```

Il est important que les utilisateurs aient accès aux type `_ fichier` en tant que type abstrait à travers une interface de module uniquement et non directement à l'aide de constructeurs du type. Dans ce dernier cas, en définissant par exemple `type _ fichier = in_channel`, le paramètre du type ne jouant aucun rôle, on pourrait tout à fait écrire une fonction de transtypage `magic: 'a fichier -> 'b fichier` qui réduirait à néant la contrainte d'accès au fichier (ouverture → lecture → fermeture) qui garantit qu'on ne peut pas lire un fichier fermé, etc.

Une implantation possible est la suivante, dans laquelle on constate que les types fantômes ne jouent aucun rôle:

```
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

```

Pour que ceci soit utile, il est nécessaire d'imposer un usage purement séquentiel d'un fichier, sinon on peut lire un nombre quelconque de caractères en parallélisant, i.e. en écrivant par exemple:

```

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

```

Le système de type ne se plaindra pas dans ce cas, ce qui constitue une faiblesse de l'approche. Néanmoins, en pratique, la plupart des codes sont naturellement séquentiels.

On peut quand même garantir cette propriété (dite de linéarité) en encapsulant l'accès au fichier par une approche monadique. Voici un exemple de code conforme "linéaire", dans lequel les deux variables successives `f` ont pour type `debut fichier` puis `fin fichier` :

```

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

```

▷ **Exercice 1** On veut étendre/adapter l'interface précédente au cas où on veut lire deux caractères, puis au cas où on veut lire un nombre pair de caractères.

▷ **Solution**

Pour lire deux caractères, on peut compter le nombre de caractères lus et s'arrêter à deux. La fonction de lecture incrémentale le nombre de caractères lus. On utilise l'encodage des entiers de Peano avec les types `zero` et `_succ`.

```

module type FichierLecture2Car =
sig
  type zero
  type _ succ
  type _ fichier
  val open_ : string -> zero fichier
  val read : 'n fichier -> char * 'n succ fichier
  val close : zero succ succ fichier -> unit
end

```

Pour lire des paires de caractères, il suffit de distinguer la parité (dans l'ordre de lecture) à l'aide de deux types distincts. Il semble alors impossible d'avoir une seule fonction de lecture, pour les caractères pairs et impairs, parce qu'elle aurait deux types différents et incompatibles. On peut tout de même éliminer cet inconvénient tout en restant dans les types fantômes, en manipulant des paires de types, que l'on échange après chaque lecture.

```

module type FichierLecturePairesDEntiers =
sig
  type even
  type odd
  type _ fichier
  val open_ : string -> (even * odd) fichier
  val readfst : ('a * 'b) fichier -> char * ('b * 'a) fichier
  val close : (even * odd) fichier -> unit
end

```

▷ **Support étudiant**

1.2 Types uniques

Les types uniques sont des types associés à une seule donnée dans tout le programme. Par ce type, on peut distinguer cette donnée de toutes les autres. Les types uniques sont généralement des types fantômes. Pour créer ce type unique (et la donnée qui va avec), on peut utiliser des modules créés dynamiquement.

```
module type TypeUnique =
sig
  type unique
  val obj : unique
end

let create_unique (type a) (v : a) =
  (module struct type unique = a let obj = v end : TypeUnique)
```

1.2.1 Application: chiffrement/déchiffrement

Considérons l'interface suivant qui correspond à un chiffrement asymétrique de type RSA. Des interfaces similaires sont disponibles en JAVA, Scala, etc.

```
module type RSA =
sig
  type key
  type secret
  (* creation d'une paire de clés publique/privée *)
  val create_key_pair : unit -> key * key
  (* chiffrement à l'aide de la clé publique *)
  val encrypt : bytes -> key -> secret
  (* déchiffrement à l'aide de la clé privée *)
  val decrypt : secret -> key -> bytes
end
```

Bien que, dans le protocole RSA, la nature des clés reste la même qu'elles soient publiques ou privées (des paires d'entiers), on pourrait les distinguer pour plus de sûreté. De même, pour ne pas mélanger les différentes paires de clés, on pourrait associer à la donnée chiffrée (le secret) la clé qui a permis de la construire.

- ▷ **Exercice 2** Modifier l'interface `RSA` pour faire apparaître et utiliser les types fantômes `pub` et `priv` pour marquer le type des clés, ainsi qu'un type unique pour identifier la paire créée.

▷ **Solution**

On aura un type `('typ, 'uniq) key` et un type `'uniq secret`. Il faut également ajouter une interface spécifiant les modules qui contiennent un type `unique` ainsi que les deux clés. La 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 clefs publique/privée *)
val create_key_pair : unit -> (module KeyPair)
(* chiffrement à l'aide de la clef publique *)
val encrypt : bytes -> (pub, 'uniq) key -> 'uniq secret
(* déchiffrement à l'aide de la clef privée *)
val decrypt : 'uniq secret -> (priv, 'uniq) key -> bytes
end

```

On propose également une solution “stub” sans aucun chiffrement, pour tester la validité de l’approche. À donner aux étudiants.

```

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
    end
    let create_key_pair () = (module
      struct
        type unique
        let pubk = ()
        let privk = ()
      end : KeyPair)

    let encrypt ba pk = ba;;
    let decrypt se pk = se;;
  end

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 on se trompe de clef*) in
  msg_decode = msg;;

```

▷ Support étudiant

1.3 Types non uniformes

Une définition récursive de type `'a t` peut faire apparaître des instances différentes du paramètre `'a`, ce qui correspondra naturellement à des fonctions récursives définies et appliquées sur ces instances différentes. De

tels types et fonctions sont appelés **non-uniformes**. Considérons par exemple le type suivant `'a perfect_tree` dans lequel on remarque que le constructeur `Node` possède un argument d'une instance différente (`'a * 'a`) `perfect_tree`. Ce type représente les “listes” contenant successivement un élément, une paire d'éléments, une paire de paires, etc. Cette description correspond également aux arbres binaires parfaitement équilibrés, si l'on considère le nombre d'éléments rangés par profondeur croissante.

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

Cet invariant structurel permet de définir des opérations pour fusionner ou couper en deux de tels arbres parfaits. Elles feront appel à la récursion non-uniforme. Pour cela, il faut annoter le type de la fonction définie et abstraire dans ce type les paramètres d'instances qui changent d'un appel à l'autre, au moyen de la notation: `type parametre. type_de_la_fonction`.

▷ **Exercice 3** Définir la fonction `split : ('a * 'a) perfect_tree -> 'a perfect_tree * 'a perfect_tree`.

▷ **Solution**

Il suffit d'annoter le type de `split` et d'abstraire le paramètre de type de `perfect_tree`. Cette implantation coupe bien l'arbre en deux sous-arbres gauches et droits, correspondants aux positions des éléments dans les paires, paires de paires, etc. Ca se passe bien car la racine est déjà une paire.

```
let rec split : type a. (a * a) perfect_tree -> a perfect_tree * a perfect_tree =
  fun arb ->
    match arb with
    | Empty -> Empty, Empty
    | Node ((t1, t2), q) -> let (q1, q2) = split q in Node (t1, q1), Node (t2, q2);;
```

▷ **Exercice 4** Quelle information (de type) manque-t-il à `'a perfect_tree` pour pouvoir définir la fonction suivante, réciproque de `split` ?

```
merge : 'a perfect_tree -> 'a perfect_tree -> ('a * 'a) perfect_tree
```

Modifier le type `'a perfect_tree` et la fonction `split` en conséquence. Définir enfin la fonction `merge`.

▷ **Solution**

Les arbres sont parfaits, mais on ignore leur profondeur, donc on ne peut savoir si deux tels arbres sont fusionnables a priori. On n'a qu'à ajouter un tel paramètre fantôme de profondeur dans le type. On peut réutiliser les types `zero` et `'a succ`. On a maintenant affaire à un GADT, avec ses propres notations de types, car les différents constructeurs engendrent des profondeurs différentes. On remarque que dans `merge`, il n'y a aucun filtrage inutile (qui donnerait `assert false`) grâce à l'annotation de profondeur, le code est donc parfaitement (et syntaxiquement) le réciproque de `split`.

```
type ('a, _) perfect_tree =
  | Empty : ('a, zero) perfect_tree
  | Node : 'a * ('a * 'a, 'p) perfect_tree -> ('a, 'p succ) perfect_tree;;

let rec split : type a p. (a * a, p) perfect_tree -> (a, p) perfect_tree * (a, p) perfect_tree =
  fun arb ->
    match arb with
    | Empty -> Empty, Empty
    | Node ((t1, t2), q) -> let (q1, q2) = split q in Node (t1, q1), Node (t2, q2);;

let rec merge : type a p. (a, p) perfect_tree -> (a, p) perfect_tree -> (a * a, p) perfect_tree =
  fun a1 a2 ->
    match a1, a2 with
```

```
| Empty      , Empty      -> Empty
| Node (t1, q1), Node (t2, q2) -> Node ((t1, t2), merge q1 q2);;
```

▷ Support étudiant

1.4 Types algébriques généralisés (GADT)

Le code suivant définit une projection “générique” des éléments d’un triplet, appelée `proj_triple`, en fonction d’un paramètre prenant trois valeurs possibles. Comme *a priori* le type des différentes projections définies par le filtrage de `proj_triple` doit être le même, on obtient comme type `proj_triple : triple_proj -> 'a * 'a * 'a -> 'a`, donc peu utilisable en pratique, alors que par exemple on a bien `first : 'a * 'b * 'c -> 'a`.

```
type triple_proj =
| First  : triple_proj
| Second : triple_proj
| Third  : triple_proj

let first (a, b, c) = a
let second (a, b, c) = b
let third (a, b, c) = c

let proj_triple = function
| First  -> first
| Second -> second
| Third  -> third
```

Pour remédier à cela, comme dans l’exercice précédent, on va définir un type `triple_proj` dont les constructeurs ont un résultat non-uniforme, i.e. un GADT. Ce paramètre non-uniforme représente le type attendu de chacune des projections. On peut maintenant définir le type “idéal” d’une projection de triplet `proj_triple` utile.

```
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
```

▷ **Exercice 5** Proposer, en généralisant le mécanisme ci-dessus, une définition de type permettant d’accéder à n’importe quelle composante de paires imbriquées. Les paires étant potentiellement imbriquées, on aura besoin d’un type récursif pour définir l’ensemble des projections possibles.

▷ Solution

Cet exercice correspond à une version typée des fonctions de la famille `cadr` présentes en LISP.

```
type _cadr_proj =
| Any    : ('a -> 'a) cadr_proj
| First  : ('a -> 'b) cadr_proj -> ('a * 'c -> 'b) cadr_proj
| Second : ('a -> 'b) cadr_proj -> ('c * 'a -> 'b) cadr_proj;;
```

```

let rec proj_cadr : type proj. proj cadr_proj -> proj =
  function
  | Any      -> (fun x      -> x)
  | First p -> (fun (a, _) -> proj_cadr p a)
  | Second p -> (fun (_, b) -> proj_cadr p b);;

(* caadr : (('a * 'b) * 'c) * 'd -> 'b *)
let caadr x = proj_cadr (First (First (Second Any))) x;;

```

En bonus, on transforme la solution précédente pour obtenir une formulation syntaxiquement proche de LISP. Ceci est facultatif. On utilise le combinateur d'application $f @@ x = (f x)$ pour éliminer le parenthésage à droite.

```

let r = Any;;
let a p = First p;;
let d p = Second p;;
let c p = proj_cadr p;;
(* le paramètre x est présent uniquement pour forcer la généralisation des types *)
let caadr x = (c @@ a @@ a @@ d @@ r) x;;

```

▷ Support étudiant

1.5 Run-Time Type Information (RTTI)

Ce mécanisme présent dans la plupart des langages orientés objet, à divers degrés, permet de représenter les types à l'exécution. Citons par exemple le `instanceof` de JAVA ou la liaison tardive qui utilisent le RTTI. On peut facilement émuler ceci à l'aide de GADT, tout en garantissant la sûreté du code. Voici un exemple de représentation de valeurs avec leur type, en l'occurrence les constantes entières et l'addition d'entiers. On construit également une fonction d'évaluation triviale:

```

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)

```

▷ Exercice 6 (Généralisons !)

On souhaite représenter et évaluer des expressions entières ne contenant que l'opération d'addition (par exemple $1+(2+3)$).

- Ajouter la possibilité a minima d'appliquer l'addition sur deux arguments entiers.
- Généraliser en permettant d'appliquer toute fonction sur tout argument (même si on ne dispose pour l'instant que de l'addition).
- Généraliser encore en permettant de représenter des valeurs de tout type.
- Généraliser enfin en permettant l'ajout de variables (question bonus).

▷ **Solution**

Pour construire des expressions entières, il suffit d'ajouter la possibilité d'appliquer l'addition sur toute paire de sous-expressions entières. D'où le constructeur `App` (pour application).

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

let plus : int repr -> int repr -> int repr =
  fun a b -> App (Add, a, b)

let un_plus_deux_plus_trois = plus (Int 1) (plus (Int 2) (Int 3))

let rec eval : type a. a repr -> a =
  function
  | Int i      -> i
  | Add       -> (fun a b -> a+b)
  | App (f, a, b) -> (eval f) (eval a) (eval b)

eval un_plus_deux_plus_trois = 6
```

Ici, l'argument fonctionnel `f` du constructeur `App` est facultatif, étant donné qu'on sait à l'avance qu'il s'agit de l'addition (seule fonction entière constructible). Néanmoins, cela permet de généraliser plus facilement, ce qui donne:

```
type _ repr =
| Int : int -> int repr
| Add : (int -> int -> int) repr
| App : ('a -> 'b) repr * 'a repr -> 'b repr

let plus : int repr -> int repr -> int repr =
  fun a b -> App (App (Add, a), b)

let un_plus_deux_plus_trois = plus (Int 1) (plus (Int 2) (Int 3))

let rec eval : type a. a repr -> a =
  function
  | Int i      -> i
  | Add       -> (fun a b -> a+b)
  | App (f, a) -> (eval f) (eval a)
```

Puis, enfin, en généralisant les valeurs:

```
type _ repr =
| Val : 'a -> 'a repr
| App : ('a -> 'b) repr * 'a repr -> 'b repr

let plus : int repr -> int repr -> int repr =
  fun a b -> App (App (Val (+), a), b)

let un_plus_deux_plus_trois = plus (Val 1) (plus (Val 2) (Val 3))

let rec eval : type a. a repr -> a =
  function
  | Val a      -> a
```

```
| App (f, a) -> (eval f) (eval a)
```

L'ajout de variables peut se faire par la gestion explicite d'un environnement et de (noms de) variables. Mais ici on peut faire plus simple car on est dans un langage fonctionnel. On va considérer que toute expression contenant par exemple une variable "x" est en fait une fonction qui, si on lui fournit la valeur de "x", permet d'évaluer complètement l'expression. Cette valeur de "x" est elle-même déterminée par une autre expression, évaluée auparavant. D'où le constructeur **Abs** (pour abstraction), l'inverse de **App**.

```
type _ repr =  
| Val : 'a -> 'a repr  
| App : ('a -> 'b) repr * 'a repr -> 'b repr  
| Abs : ('a repr -> 'b repr) -> ('a -> 'b) repr
```

```
let plus : int repr -> int repr -> int repr =  
  fun a b -> App (App (Val (+), a), b)
```

```
(* definition de l'expression 1+x *)  
let un_plus_x = Abs (fun x -> plus (Val 1) x)
```

```
let rec eval : type a. a repr -> a =  
  function  
  | Val a -> a  
  | App (f, a) -> (eval f) (eval a)  
  | Abs fr -> fun a -> eval (fr (Val a))
```

```
(* evaluation de 1+x pour x=2+3 *)  
eval (App (un_plus_x, plus (Val 2) (Val 3))) = 6
```

▷ Support étudiant

Grâce aux GADT, on peut également envisager la situation inverse des types fantômes, i.e. quand une variable de type apparaît dans les arguments d'un constructeur mais pas dans le type lui-même. Ceci correspond à un type existentiel, qui permet la génération de types uniques. Ainsi, le type ci-dessous représente-t-il une façon universelle d'encapsuler toute valeur de tout type. Néanmoins, lorsqu'on filtre une valeur du type **univ** pour récupérer la valeur fournie à la création, on ignore totalement son type, qui sera vu par le système de types, de façon sûre, comme une nouvelle constante de type unique, ce qui rend la valeur extraite complètement inutilisable ! Il faut donc avoir un vrai type abstrait, comme **stringable**, avec des primitives utiles, pour pouvoir exploiter ce mécanisme de type existentiel. On vérifie qu'on a bien défini un type existentiel car on a $\text{Any} : \forall a. a \rightarrow \text{univ} \equiv (\exists a. a) \rightarrow \text{univ}$

```
(* encapsulation universelle inutile *)  
type univ = Any : 'a -> univ  
(* encapsulation universelle des types pouvant etre convertis en chaine *)  
type stringable = Stringable : 'a * ('a -> string) -> stringable  
  
let string_of_stringable (Stringable (v, f)) = f v  
let string_of_char c = String.make 1 c  
let string_of_list f l = "[" ^ String.concat "; " (List.map f l) ^ "]"  
(* liste heterogene, contenant des int, char et int list *)  
let hliste = [Stringable (42, string_of_int);  
              Stringable ('a', string_of_char);  
              Stringable ([1;2;3], string_of_list string_of_int)]  
let string_of_hliste hl = string_of_list string_of_stringable hl  
  
string_of_hliste hliste
```

Remarque : Les types existentiels existent déjà au niveau des modules, ils correspondent simplement au mécanisme des types abstraits. Ainsi, l'implémentation par un module `Mod` d'une signature de type abstrait `Sig` contenant une déclaration `type t` équivaut à la création d'une constante de type `Mod.t` unique, dont la définition est cachée.

2 Autres constructions

Nous passons en revue succinctement d'autres constructions de types (et de valeurs) présentes dans le langage OCAML, mais nous n'étudierons pas plus ni n'exploiterons ces notions.

2.1 Types extensibles

Spécifiquement en OCAML, on peut définir des types algébriques extensibles. Cette possibilité est proche du mécanisme d'extension par sous-typage dans les langages orientés objet comme C++ ou Java.

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

Cette possibilité oblige l'utilisateur qui veut filtrer des valeurs d'un type extensible à prévoir un cas par défaut pour capturer toutes les nouvelles valeurs du type qui auraient pu être définies après.

2.2 Types variants polymorphes

Une autre possibilité d'extension de type réside dans les types variants polymorphes. Ils permettent de spécifier le type des arguments ou du retour d'une fonction comme des sous-ensembles (ou sur-ensembles) de l'ensemble des constructeurs d'un type. En contrepartie, les annotations de type et les types inférés sont plus lourds. La notion de type algébrique "clos", simple et bien défini, disparaît.

Considérons l'exemple des listes ci-dessous. On a utilisé les variants (synonymes de constructeurs) `'Nil` et `'Cons`. Ils sont dits polymorphes car on peut leur adjoindre des arguments de tous types.

Par exemple, l'expression `'Nil "coucou"` est également correcte, de type `[> 'Nil of string]`. Cette notation signifie que cette valeur est choisie dans un certain ensemble de variants (`[...]`), qui contient `'Nil` associé à un argument de type `string`, au moins (`[> ...]`). Le marqueur "au moins" permet de regrouper des valeurs de type différents dans une même structure de données contenant alors l'agrégation des différents types. Ainsi, la liste `['Nil; 'Cons (1, 'Nil)]` est de type `[> 'Nil | 'Cons of int * [> 'Nil]] list`.

La fonction `hd` a pour type `[< 'Cons of 'a * 'b] -> 'a`, ce qui signifie qu'une valeur de type `'Cons of 'a * 'b` est filtrée, au plus. On peut également annoter les fonctions (`hd` et `half`), avec éventuellement des marqueurs "au plus" ou "au moins", pour retrouver le type `'a vlist`. Il est instructif de retirer toute ou partie des annotations pour apprécier la complexité de l'inférence de types réalisée.

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

```

2.3 Types objets et classes

OCAML offre aussi un modèle de spécification et de programmation orientée objet, qui réussit à garantir la sûreté du typage en bannissant toute forme de transtypage et de tests de type dynamiques, comme on le trouve fréquemment en JAVA par exemple. Le caractère moins intuitif de ce modèle, pour qui a déjà manipulé d'autres langages à objets plus classiques, forme la contrepartie de la propriété de typage fort. On retrouve néanmoins les points principaux des langages à objets : méthodes virtuelles et privées, variable `self` et `super`, héritage (multiple), interfaces, etc. Considérons brièvement les célèbres points et points colorés:

```

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;;

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;;

```

Outre la syntaxe, la principale différence réside dans la possibilité de définir des fonctions qui s'appliqueront sur tout objet réalisant telle interface, sans avoir besoin de spécifier cette interface, comme dans la fonction:

```

val equal_x : < get_x : 'a; .. > -> < get_x : 'a; .. > -> bool

```

qui peut être spécialisée en `equal_x_point` et `equal_p1_x`.

3 Extensions

Le langage OCAML offre la possibilité d'utiliser des extensions du langage qui permettent par exemple de dériver du code automatiquement à partir de la définition d'un type. On peut ainsi obtenir "gratuitement" des itérateurs structuraux génériques. D'autres extensions permettent par exemple d'exporter/importer des structures de données OCAML au format JSON.

Il est également possible, par l'intermédiaire d'une librairie donnant accès aux structures internes du compilateur OCAML, de définir soi-même ses propres extensions. Ceci remplace un système de macros inexistant en OCAML, mais nécessite une certaine expertise de la part de l'utilisateur. Nous n'étudierons pas la définition d'extensions.