Programmation fonctionnelle

ensIIE

Semestre 3 – 2026/27

Institut Mines-Télécom
ensIIE école associée
université PARIS-SACLAY

ensIIE

www.ensiie.fr

Limites du polymorphisme

Ensembles définis de façon polymorphe

- `'a set`

Pour les ABR, utilise la comparaison et l'égalité polymorphes
`(<), (=) : 'a -> 'a -> bool`

Pour certain types, on aimerait utiliser d'autres algorithmes :

- comparer des `string` en comparant d'abord la taille plutôt que l'ordre lexicographique
- comparer des ensembles
- travailler sur des classes d'équivalence (exemple : rationnels représentés par deux entiers)

1re solution

Passer une fonction `compare : 'a -> 'a -> int` en paramètre supplémentaire de chaque fonction

`compare s t`

- 0 si `s` est égal à `t`
- < 0 si `s` est plus petit que `t`
- > 0 si `s` est plus grand que `t`

```
val add : ('a -> 'a -> int) ->
                'a -> 'a set -> 'a set
```

- lourd
- source d'erreur (mauvaise fonction passée en argument)

2e solution : type classes

Définir des classes de types qui doivent implémenter un certain nombre de fonctions

```
class Ord a where
  compare :: a -> a -> Integer
  (==)    :: a -> a -> Bool


add :: Ord a => a -> set a -> set a


instance Ord Integer where
  compare x y  =  x - y
  x == y       =  compare x y = 0
```

Pas disponible en OCaml !

3e solution : foncteurs

"Fonctions" sur les modules

Typage en utilisant les signatures

```
(* .mli *)
module M (X : S) : S'
(* ou *)
module M : functor (X : S) -> S'

(* .ml *)
module M(X : S) = struct ... end
(* ou *)
module M = functor (X : S) -> struct ... end
```

Pour les ensembles

```
module type OrderedType = sig
  type t
  val compare : t -> t -> int
end

module Make(O : OrderedType) = struct
  ...
end

module StringSet = Make(String)
```