Programmation Fonctionnelle Avancée

Séance 3 : Types abstraits, includes, contraintes de type, foncteurs

Alexandros Singh

Université Paris 8

23 octobre 2023

Définition

Un ensemble (set) est une collection non ordonnée d'éléments distincts.

Voici un exemple de signature pour un module `Set` (voir `https://cs3110.github.io/textbook/chapters/modules/functional_data_structures.html`) :

```
module type Set = sig
  type 'a t
  val empty : 'a t
  val mem : 'a -> 'a t -> bool
  val add : 'a -> 'a t -> 'a t
  val elements : 'a t -> 'a list
end
```

Le type `'a t` n'est pas défini dans cette signature, c'est un type abstrait.

Types abstraits

Voici un exemple d'implémentation à l'aide de listes :

```
module UniqListSet : Set = struct
  type 'a t = 'a list
  let empty = []
  let mem = List.mem
  let add x s = if mem x s then s else x :: s
  let elements s = Fun.id
  end
```

Bien que le code ici fonctionne parfaitement, le toplevel ne sait pas afficher les valeurs du type abstrait `'a t` :

```
# UniqListSet.(add 5 empty |> add 10 |> add 20 |> add 10);;
- : int UniqListSet.t = <abstr>
```

Pour y remédier, nous pouvons étendre notre définition pour y inclure une fonction d'impression "pretty-printing" :

```
module type Set = sig
  type 'a t
  val empty : 'a t
  val mem : 'a -> 'a t -> bool
  val add : 'a -> 'a t -> 'a t
  val elements : 'a t -> 'a list
  val pp :
    (Format.formatter -> 'a -> unit) ->
    Format.formatter -> 'a t -> unit
end
```

Et voici la nouvelle implémentation utilisant des listes :

```
module UniqListSet : Set = struct
  type 'a t = 'a list
  let empty = []
  let mem = List.mem
  let add x s = if mem x s then s else x :: s
  let elements = Fun.id
  let pp pp_val fmt s =
    let open Format in
    fprintf fmt "{";
    pp_print_list ~pp_sep:(fun out () -> fprintf out ", ")
                  pp_val fmt s;
    fprintf fmt "}";
end
```

Voir `https://v2.ocaml.org/api/Format.html` pour plus de détails !

Nous pouvons maintenant utiliser cette fonctionnalite comme suit :

```
(* demande au toplevel d'utiliser notre pretty-printer lorsqu'il doit
imprimer des valeurs de type UniqListSet.t *)
# #install_printer UniqListSet.pp;;
# UniqListSet.(add 5 empty |> add 10 |> add 20 |> add 10);;
- : int UniqListSet.t = {20, 10, 5}
```

Includes

Supposons maintenant que nous voulions étendre notre signature pour les ensembles afin d'y inclure de nouvelles fonctionnalités. Nous pourrions procéder comme précédemment et modifier simplement la signature originale. Mais que se passe-t-il si nous n'avons pas accès à la signature et au module d'origine (peut-être font-ils partie d'une bibliothèque) ? Dans ce cas, nous définissons une nouvelle signature et `include` l'ancienne :

```
module type SetExtended = sig
    include Set
    val of_list : 'a list -> 'a t
    val map : ('a -> 'b) -> 'a t -> 'b t
end
```

Includes

Nous pouvons également utiliser `include` à l'intérieur d'un module pour inclure toutes les définitions d'un autre module. Nous pouvons donc définir un module conforme à la signature SetExtended comme suit :

```
module UniqListSetExtended : SetExtended = struct
    include UniqListSet
    let of_list lst = List.fold_right add lst empty
    let map f s = List.map f (elements s) |> of_list
end
```

```
# UniqListSetExtended.(add 5 empty |> add 10 |> add 20 |> add 10 |> map
↪  (fun x -> 2*x));;
- : int UniqListSetExtended.t = {40, 20, 10}
```

Mais attendez, puisque dans `UniqListSet` nous avons `'a t = 'a list`, pourquoi n'avons-nous pas simplement écrit :

```
module UniqListSetExtended : SetExtended = struct
    include UniqListSet
    let of_list lst = lst
    let map f s = List.map f s
end
```

En essayant de charger ceci dans le toplevel, nous sommes confrontés à un message d'erreur :

```
Error: Signature mismatch:
       ...
       Values do not match:
         val of_list : 'a -> 'a
       is not included in
         val of_list : 'a list -> 'a t
       The type 'a list -> 'a list is not compatible with the type
         'a list -> 'a t
       Type 'a list is not compatible with type 'a t
```

Pourquoi ?

Includes

En essayant de charger ceci dans le toplevel, nous sommes confrontés à un message d'erreur :

```
Error: Signature mismatch:
       ...
       Values do not match:
         val of_list : 'a -> 'a
       is not included in
         val of_list : 'a list -> 'a t
       The type 'a list -> 'a list is not compatible with the type
         'a list -> 'a t
       Type 'a list is not compatible with type 'a t
```

Encapsulation : puisque nous avons déclaré `UniqListSet : Set`, les détails de l'implémentation de `UniqListSet.t` sont cachés - tout ce qui reste est le type abstrait !

Par contre, cela fonctionne :

```
module UniqListSetImpl = struct
  type 'a t = 'a list
  let empty = []
  let mem = List.mem
  let add x s = if mem x s then s else x :: s
  let elements = Fun.id
  let pp pp_val fmt s =
    let open Format in
    fprintf fmt "{";
    pp_print_list ~pp_sep:(fun out () -> fprintf out ", ") pp_val fmt s;
    fprintf fmt "}";
end

module UniqListSet : Set = UniqListSetImpl

module UniqListSetExtended : SetExtended = struct
    include UniqListSetImpl
    let of_list lst = lst
    let map f s = List.map f s
end
```

Includes

Par contre, cela fonctionne :

```
module UniqListSetImpl = struct
  type 'a t = 'a list
  let empty = []
  let mem = List.mem
  let add x s = if mem x s then s else x :: s
  let elements = Fun.id
  let pp pp_val fmt s =
    let open Format in
    fprintf fmt "{";
    pp_print_list ~pp_sep:(fun out () -> fprintf out ", ") pp_val fmt s;
    fprintf fmt "}";
end

module UniqListSet : Set = UniqListSetImpl

module UniqListSetExtended : SetExtended = struct
    include UniqListSetImpl
    let of_list lst = lst
    let map f s = List.map f s
end
```

Comme nous n'avons pas défini de signature pour le module `UniqListSetImpl`, une signature est déduite automatiquement, dans laquelle `'a t = 'a list` est valable.

Définition

Un monoïde est un ensemble équipé d'une opération binaire associative et d'un élément identité.

Voici une signature de monoïdes en OCaml :

```
1 module type Monoid = sig
2   type t
3   val i : t
4   val ( @ ) : t -> t -> t
5 end
```

Type constraints

L'ensemble des entiers munis de l'opération d'addition est un monoïde :

```
module IntegersUnderAddition : Monoid = struct
  type t = int
  let i = 0
  let ( @ ) = Stdlib.( + )
end
```

Une fois de plus, `IntegersWithAddition.t` apparaît comme un type abstrait. Donc, par exemple, cette expression (qui s'évalue à `0 + 0 = 0`) donne :

```
# IntegersUnderAddition.(i @ i);;
- : IntegersUnderAddition.t = <abstr>
# IntegersUnderAddition.(i @ i = i);;
- : bool = true
```

Type constraints

Le problème de l'affichage des valeurs du type `IntegersWithAddition.t` sous la forme `<abstr>` peut en effet être résolu à l'aide d'une jolie imprimante, mais le problème le plus important est le suivant :

```
# IntegersUnderAddition.(i @ i = 0);;
Error: This expression has type int but an expression was expected of type
↪  t
# IntegersUnderAddition.(i @ 5);;
Error: This expression has type int but an expression was expected of type
↪  t
```

Puisque l'information que `IntegersWithAddition.t = int` n'est pas exposée au toplevel, il ne nous permettra pas de mélanger les valeurs de notre monoïde avec des entiers !

Type constraints

OCaml nous permet de spécialiser les signatures en introduisant des contraintes qui sont exposées à tous :

```
1 module type IntMonoid = Monoid with type t = int
```

Qui est évaluée à :

```
module type IntMonoid = sig
    type t = int
    val i : t
    val ( @ ) : t -> t -> t
end
```

Type constraints

En remplaçant la signature de notre implémentation de monoïde par cette nouvelle signature que nous avons maintenant :

```
module IntegersUnderAddition : IntMonoid = struct
  type t = int
  let i = 0
  let ( @ ) = Stdlib.( + )
end
```

Et puisque le fait que `IntegersWithAddition.t = int` est maintenant une information publique, nous avons :

```
utop # IntegersUnderAddition.(i @ 5);;
- : int = 5
utop # IntegersUnderAddition.(i @ i = 0);;
- : bool = true
```

- Et si nous voulions tester si les modules que nous avons définis sont bien des monoïdes au sens mathématique ?
- La signature que nous avons écrite ne vérifie pas si la valeur `i` agit comme l'identité de `@`, ni si `@` est associatif !
- Nous avons donc besoin d'une fonction qui prend en entrée un module de signature `IntMonoïde` et l'utilise pour évaluer un certain nombre d'expressions pour tenter de vérifier les axiomes des monoïdes.

En OCaml, les foncteurs sont exactement ça : des fonctions entre modules !

```
module MonoidTester (M : IntMonoid) = struct
    let test_identity =
        List.fold_right (fun x y -> (M.(@) M.i x = x) && (M.(@) x M.i = x) && y) [1;2;3;4;5;-1;-2;-3;-4;-5] true
    let test_associativity =
        let helper  = function x,y,z -> (M.(@) (M.(@) x y) z) = (M.(@) x (M.(@) y z)) in
        List.fold_right (fun x y -> (helper x) && y) [(1,2,3);(5,-10,15);(150,-5,30)] true
end
```

```
module MonoidTester :
 functor (M : Code.Monoids.IntMonoid) ->
   sig val test_identity : bool val test_associativity : bool end
```

Foncteurs

Définissons deux autres monoïdes à tester :

```
module IntegersUnderMultiplication : IntMonoid = struct
  type t = int
  let i = 1
  let ( @ ) = Stdlib.( * )
end

module FalseMonoid : IntMonoid = struct
  type t = int
  let i = 42
  let ( @ ) = Stdlib.( + )
end

module TestAdd = MonoidTester (IntegersUnderAddition)
module TestMul = MonoidTester (IntegersUnderMultiplication)
module TestFalse = MonoidTester (FalseMonoid)
```

Enfin, l'évaluation nous donne :

```
utop # TestAdd.test_identity;;
- : bool = true
utop # TestAdd.test_associativity;;
- : bool = true
utop # TestMul.test_identity;;
- : bool = true
utop # test_associativity;;
- : bool = true
utop #TestFalse.test_identity;;
- : bool = false
utop # TestFalse.test_associativity;;
- : bool = true
```

Nous avons détecté que `FalseMonoid` n'est en effet pas un monoïde !