Cours 4 – Programmation générique et modulaire

Programmation fonctionnelle

CFA INSTA - Master 1 - Analyste Développeur

Pierre Talbot (pierre.talbot@univ-nantes.fr)

Université de Nantes

16 avril 2019

UNIVERSITÉ DE NANTES

Le menu

- Polymorphisme
- Modules
- Foncteurs

Types Polymorphiques

$\langle T, U, \ldots \rangle$ ::=	**Types** Ty_4
| ...	(types Ty_3)
| **'a**, **'b**, ...	(type polymorphe)
$\langle D \rangle$ ::=	**Déclaration de types** D_4
| ...	(déclarations D_3)
| `type 'a t =` T	(alias polymorphe)
| `type 'a t =` `{` **x** : T `;` **y** : U `}`	(enregistrement polymorphe)
| `type 'a t =` **VarT** `of` T | **VarU** `of` U	(type somme polymorphe)

- Note : Les paramètres de types polymorphiques sont n-aires.

Polymorphisme

On peut généraliser l'exemple de la structure de liste d'entier :

```
type int_list =
| Empty
| Value of (int * int_list)
```

pour qu'elle puisse transporter n'importe quel type :

```
type 'a list =
| Empty
| Value of ('a * 'a list)
```

Les types précédés d'une apostrophe `'a` sont des *types polymorphiques*.

Polymorphisme

Les fonctions peuvent également avoir un type polymorphique, par exemple :

```
(fun x -> x) : 'a -> 'a
```

On ne peut pas inférer d'information sur x et il n'y a pas de conflit dans son utilisation, donc on considère que c'est un type polymorphique.

Le menu

- Polymorphisme
- Modules
- Foncteurs

Modules OCaml

Un langage de module “au-dessus” de OCaml, très utile et puissant :

- Permet d'écrire des **abstractions** : isoler des composants d'un programme en unités distinctes.
- Un **module** implémente une **signature** décrivant l'interface publique du module.
- Un **foncteur** permet de programmer un module par un autre module.

Langage de modules

$\langle S \rangle$ ::=	**Module signature**
| **ASig**, **BSig**, ...	(nom de signature)
| `sig` D `end`	(signature)
| `functor (`**A** : S `) ->` S	(functor)
$\langle M \rangle$ ::=	**Langage de module**
| **A**, **B**, ...	(nom de module)
| `struct` D `end : ` S	(module)
| `functor (`**A** : S `) ->` M	(functor)
$\langle D \rangle$ ::=	**Déclaration de module** D_5
| ...	(déclarations D_4)
| `module` **A** : S = M	(Déclaration module)
| `module type` **ASig** = S	(Déclaration signature)
|	

Un module de liste

```
module List =
struct
  type 'a t = Empty | Value of ('a * 'a t)
  let cons x l = Value (x, l)
end
```

L'utilisateur n'a pas besoin de connaitre l'implémentation ni le type exact de la liste :

```
module type ListSig =
sig
  type 'a t
  val cons : 'a -> 'a t -> 'a t
end
```

Annoter le type d'un module

On peut annoter un module avec une signature comme cela :

```
module List : ListSig =
struct
  type 'a t = Empty | Value of ('a * 'a t)
  let cons x l = Value (x, l)
end
```

ou même directement :

```
module List : sig
  type 'a t
  val cons : 'a -> 'a t -> 'a t
end =
struct
  ...
end
```

Tout fichier est un module

Si un fichier `a.ml` contient

```
let plus a b = a + b
```

alors automatiquement le compilateur va créer un module `A` :

```
module A =
struct
  let plus a b = a + b
end
```

et sa signature :

```
module type A =
sig
  val plus : int -> int
end
```

Bonne pratique : Séparer implémentation et abstraction

Une bonne pratique est de séparer le code des modules et des signatures dans deux fichiers différents :

- `list.ml` : Implémentation du module de liste.
- `list.mli` : Seulement la signature du module liste.

Le menu

- Polymorphisme
- Modules
- Foncteurs

Foncteur

- Un foncteur permet de paramétriser un module avec un autre module.
- Cela permet d'écrire du code **générique** qui est réutilisé dans plusieurs contexte.

Exemple

- On veut faire un module qui fournit des fonctions mathématiques indifféremment du type `int` et `float`.
- On encapsule ces deux types dans un module avec une fonction `plus` :

```
module Integer = struct
  type t = int
  let zero = 0
  let plus = + end

module Float = struct
  type t = float
  let zero = 0.
  let plus = +. end
```

Signature commune

Les types `Integer` et `Float` ont une même signature :

```
module type NumberSig = sig
  type t
  val zero : t
  val plus : t -> t -> t
end
```

On voudrait écrire un foncteur qui utilise cette signature et permet d'écrire une fonction `sum`.

Foncteur pour une fonction somme n-aires

On peut utiliser les fonctions du module paramétré `N` pour programmer la fonction somme :

```
module type ArithSig = functor(N : Number) ->
sig
  val sum : N.t list -> N.t
end

module Arith : ArithSig = functor(N : Number) ->
struct
  let sum l = List.fold_left N.plus N.zero l
end
```

Utilisation d'un foncteur

On peut instancier le foncteur `Arith` avec le module `Integer` et l'utiliser dans une fonction.

```
module ArithInt = Arith(Integer)
let () =
  Printf.printf "%d\n" (ArithInt.sum [1;4;43])
```