Programmation fonctionnelle avec OCaml

5ème séance, 16 avril 2015

Le langage des modules

samuel.hornus@inria.fr

http://www.loria.fr/~shornus/ocaml/

Rappels de la 3ème séance

```
module type PILE =
  sig
    type 'a t
    val create : unit -> 'a t
    val push : 'a -> 'a t -> unit
    val pop : 'a t -> 'a
  end
module Pile : PILE =
  struct
    type 'a t = 'a list ref
    let create () = ref []
    let push x p = p := x::!p
    let pop p = match !p with [...]
    let rec print p = match  p with [...]
  end
```

Rappels de la 3ème séance

signature de module

```
module type PILE =
  sig
    type 'a t
    val create : unit -> 'a t
    val push : 'a -> 'a t -> unit
    val pop : 'a t -> 'a
  end
module Pile : PILE =
  struct
    type 'a t = 'a list ref
    let create () = ref []
    let push x p = p := x::!p
    let pop p = match !p with [...]
    let rec print p = match  p with [...]
  end
```

Rappels de la 3ème séance

signature de module

```
module type PILE =
  sig
    type 'a t
    val create : unit -> 'a t
    val push : 'a -> 'a t -> unit
    val pop : 'a t -> 'a
  end
```

module *restreint* par la signature PILE

```
module Pile : PILE =
  struct
    type 'a t = 'a list ref
    let create () = ref []
    let push x p = p := x::!p
    let pop p = match !p with [...]
    let rec print p = match p with [...]
  end
```

Rappels de la 3ème séance

- `struct ... end` introduit une collection de définitions : valeurs, types ou modules. C'est une structure.
- `module` *`Nom`* `= struct ... end` permet de donner un nom à cette structure et en fait un *module*.

```
module Pile : PILE =
  struct
    type 'a t = 'a list ref
    let create () = ref []
    let push x p = p := x::!p
    let pop p = match !p with [...]
    let rec print p = match p with [...]
  end
```

module *restreint* par la signature `PILE`

Rappels de la 3ème séance

signature de module

```
module type PILE =
  sig
    type 'a t
    val create : unit -> 'a t
    val push : 'a -> 'a t -> unit
    val pop : 'a t -> 'a
  end
```

- `sig ... end` introduit une *signature de module* : une *interface* pour un module.
- `module type` *NOM* `= sig ... end` permet de donner un nom à cette signature.
- On *restreint* souvent une structure par une signature pour « cacher » certaines définitions : `module Pile : PILE = ...`

Utilisation d'un module

En ≪ dehors ≫ du module, on accède à ses composants grâce à la notation pointée :

```
# let p = Pile.create ();;
val p : '_a Pile.t = <abstr>
# Pile.push 45 p;;
- : unit = ()
# Pile.pop p;;
- : int = 45
# Pile.pop p;;
Exception: Failure "erreur : la pile est vide".
```

Utilisation d'un module

En « dehors » du module, on accède à ses composants grâce à la notation pointée :

```
# let p = Pile.create ();;
val p : '_a Pile.t = <abstr>
# Pile.push 45 p;;
- : unit = ()
# Pile.pop p;;
- : int = 45
# Pile.pop p;;
Exception: Failure "erreur : la pile est vide".
```

Une *signature de module* fournit une interface entre l'intérieur et l'extérieur du module.
Elle permet de cacher des définitions et de restreindre (ou abstraire) des types.

Le langage des modules

```
module type NomDeLaSignature =
  sig
    type ...
    val nom : type
    module Nom : Signature
    ...
  end
```

signature

≈ type

Le langage des modules

```
module type NomDeLaSignature =
  sig
    type ...
    val nom : type
    module Nom : Signature
    ...
  end
```

signature

≈ type

```
module NomDuModule [: Signature]
  struct
    type nom = ...
    let nom [paramètres...] = ...
    module Nom : Signature
    ...
  end
```

module/structure

≈ valeur

Le langage des modules : les foncteurs

```
module Nom =
  functor (M1:S1) -> functor (M2:S2) -> ... ->
struct
    ...
end
```

foncteur

$\approx$ fonction

foncteur = « fonction » d'une structure vers une structure.

foncteur = « module paramétré » par d'autres modules

Le langage des modules : les foncteurs

```
module Nom =
  functor (M1:S1) -> functor (M2:S2) -> ... ->
struct
    ...
end
```

foncteur

$\approx$ fonction

foncteur = « fonction » d'une structure vers une structure.
foncteur = « module paramétré » par d'autres modules

Autre syntaxe :

```
module Nom (M1:S1) (M2:S2) (M3:S3) ... =
struct
    ...
end
```

Le langage des modules : les foncteurs

Syntaxe d'une signature d'un foncteur :

```
module type Nom =
  functor (M1:S1) -> functor (M2:S2) -> ... ->
sig
    ...
end
```

Restriction d'un module par une signature

Sert à *abstraire* certains types et *cacher* certaines valeurs (données, fonctions) :

```
module Nom : Signature = struct ... end
```

```
module Nom = Nom : Signature
```

Sert à spécifier le contenu des modules paramètres d'un foncteur :

```
module Nom : Signature =
    functor (M1 :S1) -> functor (M2 :S2) -> ...
    struct ... end
```

DÉMO (ex1)

Application d'un foncteur

Étant donné

```
module F (P1:S1) (P2:S2) ... (Pn:Sn)
= struct ... end
```

On *applique* un foncteur à des paramètres modules, pour obtenir un nouveau module :

```
module M = F (Titi) (Toto) ...
```

`M` est le nouveau module, résultat du foncteur `F` appliqué aux n modules `Titi`, `Toto`, ...

`Titi` doit respecter la signature `S1` ; `Toto` doit respecter la signature `S2`, etc.

DÉMO (ex2 et ex3)

Codons un module

On veut créer un module pour gérer des ensembles de valeurs.
On veut :

- créer un ensemble vide
- ajouter des valeurs dans un ensemble (du même type)
- savoir si une valeur est dans un ensemble
- supprimer une valeur d'un ensemble

Codons un module

On veut créer un module pour gérer des ensembles de valeurs.
On veut :

- créer un ensemble vide
- ajouter des valeurs dans un ensemble (du même type)
- savoir si une valeur est dans un ensemble
- supprimer une valeur d'un ensemble

Ajout d'une égalité de type :

```
(Signature with type t1 = t2)
```

où *t1* est un type déclaré dans *Signature*

Exemple : protocoles de communication

http ://www.cs.cmu.edu/Groups/fox/papers/lfp-signatures.ps

```
module type PROTOCOL = sig ... end
module Tcp (Lower : PROTOCOL)= struct ... end
module Ip (Lower : PROTOCOL) = struct ... end
module Ethernet (Lower : PROTOCOL) = struct ... end
```

Instanciation :

```
module EthDevice : DEVICE_PROTOCOL = struct ... end
module EthInstance = Eth(EthDevice)
module IpInstance = Ip(EthInstance)
module TcpInstance = Tcp(IpInstance)
```

Utilisation :

```
let connexion = TcpInstance.active_open ...
```