B4- Functional Programming

B-PAV-360

Bootstrap : Functors

Camels are far too intelligent to admit to being intelligent.

1.0

Bootstrap : Functors

Camels are far too intelligent to admit to being intelligent.

Ce bootstrap de programmation fonctionelle avec OCaml vous familiarisera avec les functors. Nous manipulerons dans un premier temps les functors comme un outil de génération de code puis nous découvrirons les functors de la bibliothèque standard du langage.

Pour ce dernier boostrap, vous n'avez pas de préliminaires à préparer avant la session.

Écrire des functors

Exercice 1

Pour dédramatiser un peu les functors, nous allons prendre un exemple proche de celui du cours. Pour rendre cet exemple un peu plus utile, vous allez écrire un functor similaire, mais qui prendra deux modules en paramètres au lieu d'un seul et renverra un module contenant un couple des valeurs de ses deux paramètres.
Pour faire un parallèle avec des fonctions, l'esprit est le suivant :

```
let couple_cours = fun x -> (x, x)
let couple_tp    = fun x -> fun y -> (x, y)
```

Allons-y !

- Écrivez la signature `VAL` des modules contenant au moins une valeur `v` de type entier.
- Écrivez la signature `COUPLE` des modules contenant au moins une valeur `couple` de type `int * int`.
- Écrivez la signature `MAKECOUPLE` des functors prenant deux paramètres de signature `VAL` et renvoyant un module de signature `COUPLE`.
- Écrivez un functor `MakeCouple` de signature `MAKECOUPLE` qui associe à la valeur `couple` de type `int * int` un tuple à deux éléments, avec pour premier élément la valeur `v` de son premier paramètre et pour second élément la valeur `v` de son second paramètre.
- Générez quelques modules de signature `COUPLE` avec votre functor `MakeCouple`.

Exercice 2

Nous allons maintenant faire le travail inverse. Un couple peut être déconstruit de deux façons : Soit on recupère le premier élément, soit on récupere le second. Pour faire de nouveau un parallèle avec les fonctions, c'est ce que font les fonctions de projection de la bibliotèque standard fst et snd définies de la façon suivante :

```
let fst = fun (x, y) -> x
let snd = fun (x, y) -> y
```

Allons-y !

- Écrivez une signature `PROJECTION` des functors prenant un seul paramètre de signature `COUPLE` et renvoyant un module de signature `VAL`.
- Écrivez un functor `Fst` de signature `PROJECTION` qui associe le premier élément de la valeur `couple` de son paramètre à l'entier `x` du module qu'il génère.
- Écrivez un functor `Snd` de signature `PROJECTION` qui associe le second élément de la valeur `couple` de son paramètre à l'entier `x` du module qu'il génère.
- Générez quelques modules de signature `VAL` avec vos functors `Fst` et `Snd`.

Exercice 3

Vous allez écrire un générateur d'évaluateur d'expressions. Oui oui, il est bien question de coder un meta-evalexpr.

Pas de panique, c'est loin d'etre aussi compliqué que le nom le laisse entendre. A l'heure actuelle, vous avez probablement écrit des dizaines d'evalexpr avec tous les langages que vous connaissez. Leur forme vous est donc familière. La valeur ajoutée de cet exercice est de vous faire manipuler les types abstraits avec les functors. Nous allons considérer un evalexpr très simple en nous limitant à trois formes d'expressions :

- Les additions
- Les multiplications
- Les valeurs immediates

Voici le code qui nous servira de base de réflexion :

```
module EvalExpr =
struct
  type expr =
    | Add of (expr * expr)
    | Mul of (expr * expr)
    | Value of int

  let rec eval = function
    | Add (lhs, rhs) -> ( + ) (eval lhs) (eval rhs)
    | Mul (lhs, rhs) -> ( * ) (eval lhs) (eval rhs)
    | Value v        -> v
end
```

Et la signature associée :

```
module type EVALEXPR =
sig
  type expr =
    | Add of (expr * expr)
    | Mul of (expr * expr)
    | Value of int

  val eval : expr -> int
end
```

Si notre meta-evalexpr doit générer des evalexprs, ça veut avant tout dire que les types manipulés DOIVENT supporter les operations d'addition ET de multiplication. C'est donc une bonne idée de commencer par traiter cette problématique.

- Écrivez la signature de module `VAL` des modules possédant au moins un type abstrait `t`, une fonction `add : t -> t -> t` et une fonction `mul : t -> t -> t`.
- Écrivez les modules `IntVal` et `FloatVal` respectant cette signature, MAIS NE LA LIEZ PAS EXPLICITEMENT.

Si vous liez explicitement `IntVal` et `FloatVal` à la signature `VAL`, il ne sera plus possible de connaître la nature réelle du type t associé. C'est pour cela qu'en ne liant pas ces modules à la signature `VAL`, le compilateur leur infère une signature où le type `t` est concret qui est une signature compatible avec la signature `VAL`.

Vous devriez maintenant pouvoir écrire des expressions de ce genre :

```
# IntVal.add 40 2;;
- : int = 42
# FloatVal.add 41.5 0.92;;
- : float = 42.42
```

Maintenant que nous avons une bonne idée ce que notre functor de meta-evalexpr va prendre en paramètre, il est temps de nous intéresser à ce qu'il va générer : des evalexpr. Ces evalexpr vont bien entendu respecter une signature précise et proche de celle présentée plus haut. Ils ne manipuleront bien évidemment pas le type `int` mais un type quelconque `t`.

Écrivez la signature de module `EVALEXPR` des modules possédant au moins :

- Un type abstrait t
- Un type concret expr qui est un variant à 3 constructeurs similaires à l'exemple du début de l'exercice (attention à la valeur construite par Value !)
- Une fonction `eval : expr -> ???` dont vous devez deviner le type de retour.

Nous avons maintenant une signature pour les modules pris en paramètres par notre functor (le meta-evalexpr) et pour les modules générés par celui-ci. Il est maintenant temps d'écrire la signature de ce functor !

Écrivez la signature `MAKEEVALEXPR` des functors prenant un module de signature `VAL` en paramètre et renvoyant un module de signature `EVALEXPR`.

Attention !!! le type `t` des evalexpr générés est abstrait ! Il est donc obligatoire de lier explicitement ce type avec le type `t` du module pris en paramètre ! Vous devez utiliser pour cela la syntaxe "`with type ... = ...`" à la fin de votre signature de functor. Si vous n'êtes pas très à l'aise, revoyez la dernière partie du cours.

Dernière étape, le functor ! Écrivez le functor `MakeEvalExpr` de signature `MAKEEVALEXPR` qui génère un evalexpr en fonction du type `t` et des fonctions `add` et `mul` du module de signature `VAL` passé en paramètre.

Félicitations ! Vous voilà les heureux propriétaires d'un meta-evalexpr ! Vous devriez donc maintenant pouvoir écrire des expressions de la forme :

```
module IntEvalExpr = MakeEvalExpr (IntVal)
module FloatEvalExpr = MakeEvalExpr (FloatVal)
```

Amusez-vous à tester vos evalexpr avec des expressions du genre :

```
IntEvalExpr.eval (IntEvalExpr.Add
(IntEvalExpr.Value 40, IntEvalExpr.Value 2))
```

ou encore :

```
FloatEvalExpr.eval (FloatEvalExpr.Add
(FloatEvalExpr.Value 41.5, FloatEvalExpr.Value 0.92))
```