Université Paris Diderot
M1 Informatique

Programmation Fonctionnelle Avancée
Année 2012-2013

TP n°1 - Correction

Modules

Les exercices 1,2,3,6 de ce TP sont extraits du cours de Didier Rémy disponible à l'adresse `http://gallium.inria.fr/~remy/isia`.

1 Conversions entre monnaies

Exercice 1 [Monnaies] L'Euro et le Dollar sont deux monnaies supportant les mêmes opérations mais incompatibles entre elles.

1. Définissez un module Mfloat contenant un type `t = float`, ainsi que les opérations `un`, `plus`, et `prod` permettant de créer l'unité pour le type `t`, d'additionner deux éléments de type `t`, et de faire leur produit.
2. Écrivez une interface `MONNAIE` de telle façon à ce que le type `t` soit abstrait.
3. On peut maintenant créer deux modules `Euro` et `Dollar` par le code suivant :

   ```
   module Euro = (MFloat : MONNAIE);;
   module Dollar = (MFloat : MONNAIE);;
   ```

   Vérifiez que les deux monnaies sont bien incompatibles entre elles. Quel mécanisme de OCaml permet de s'en assurer ?
4. Ecrivez une fonction `euro : float -> Euro.t` qui fabrique des Euros.

Correction :

```
module type MONNAIE =
  sig
    type t
    val un  : t
    val plus : t -> t -> t
    val prod : float -> t -> t
  end
;;

module MFloat =
  struct
    type t = float
    let un = 1.0
    let plus = (+.)
    let prod = ( *. )
  end
;;

module Euro = (MFloat : MONNAIE);;
module Dollar = (MFloat : MONNAIE);;
```

```
let euro x = Euro.prod x Euro.un;;
Euro.plus (euro 10.0) (euro 20.0);;

(* incompatibilité : Euro.plus (euro 50.0) Dollar.un;; *)
```

Exercice 2 [Bureau de change] On souhaite maintenant créer un bureau de change sous forme de module qui permette de convertir une somme en euros en une somme en dollars.

1. Donnez la signature du module la plus abstraite (par exemple le taux de change n'a pas besoin d'être connu). Rappel : la signature d'un module peut elle même contenir des variables de type module.
2. Donnez en une implémentation.

Correction :

```
module type BUREAU_DE_CHANGE =
  sig
    module Euro : MONNAIE
    module Dollar : MONNAIE
    val euro_to_dollar : Euro.t -> Dollar.t
    val dollar_to_euro : Dollar.t -> Euro.t
  end
;;

module BureauDeChange : BUREAU_DE_CHANGE =
  struct
    module Euro = MFloat
    module Dollar = MFloat
    let un_euro_en_dollar = 0.95
    let euro_to_dollar x = x /. un_euro_en_dollar
    let dollar_to_euro x = x *. un_euro_en_dollar
  end
;;

let deux_euro = BureauDeChange.Euro.prod 2.0 BureauDeChange.Euro.un;;
let deux_euro_en_dollar = BureauDeChange.euro_to_dollar deux_euro;;
```

Exercice 3 [Restrictions] On souhaite définir un module qui permette uniquement d'additionner des euros mais qu'il soit en revanche impossible d'en créer ou d'en multiplier.

1. Donnez une signature `MPLUS` qui permette cette restriction.
2. A partir de cette signature et du module `Euro` créé précédemment, créez un module `Mplus` de la façon la plus concise possible.
3. Vérifiez que votre code fonctionne i.e. :
   (a) L'expression suivante doit être bien typée :
   ```
   MPlus.plus Euro.un Euro.un;;
   ```
   (b) Mais l'expression suivante doit retourner une erreur :
   ```
   MPlus.plus Euro.un 1.0;;
   ```

Correction :

```
module type MPLUS =
  sig
    type t
    val plus : t -> t -> t
  end
;;

module Mplus = (Euro : MPLUS with type t = Euro.t);;
Mplus.plus Euro.un Euro.un;;
(* Par contre ce code donne erreur *)
(* MPlus.plus Euro.un 1.0;; *)
```

2 Encore la compilation séparée

Exercice 4 Reprenez le dernier exercice du TP précedent, et constuisez un fichier source unique contenant des définitions de modules et interfaces de modules correspondantes aux fichiers `plus.ml`, `fois.ml`, `exp.ml` et `main.ml`.

3 Intermède informatique

Exercice 5 On définit le module `Ordi` de la façon suivante :

```
module Ordi =
struct
  type t = int ref
  let create () = ref 0
  let start s = s:=1
  let read_state s = !s
end
```

Donnez deux signatures `USER_ORDI` et `ADMIN_ORDI` de telle façon à ce que le type `t` soit abstrait, et que seul l'administrateur puisse accéder aux fonctions `start` et `create`.

L'utilisateur pourra seulement utiliser la fonction `read_state`.

Correction :

```
module Ordi =
struct
  type t = int ref
  let create () = ref 0
  let start s = s:=1
  let read_state s = !s
end

module type USER_ORDI =
sig
  type t
  val read_state : t -> int
end

module type ADMIN_ORDI =
sig
```

```
  type t
  val create : unit -> t
  val start : t -> unit
  val read_state : t -> int
end

module UserOrdi = (Ordi : USER_ORDI with type t = Ordi.t)
module AdminOrdi = (Ordi : ADMIN_ORDI with type t = Ordi.t)

let pc1 = AdminOrdi.create ();;
UserOrdi.read_state pc1;;
AdminOrdi.start pc1;;
UserOrdi.read_state pc1;;
```

Exercice 6 Voici un extrait du code de Menhir, un générateur de parseurs pour OCaml :

```
(* A uniform interface for output channels. *)
module type OUTPUT = sig
  type channel
  val char: channel -> char -> unit (* Ecrit un caractère sur le canal *)
  val substring: channel -> string -> int (* offset *) -> int (* length *) -> unit
(* Ecrit length caractères de la chaine string à partir de l'offset
sur le canal channel *)
end
```

On souhaite donner deux implémentations de cette interface : une fonctionnant avec l'entrée standard, l'autre avec le module `Buffer`. A partir de la documentation du module `Buffer` et du module `Pervasives` (le module chargé par défaut qui contient les informations sur les entrées/sorties standards), écrivez `ChannelOutput` qui a comme type concret la sortie standard, et `BufferOutput` qui a comme type concret le type abstrait des buffers.
Correction :

```
module ChannelOutput : OUTPUT with type channel = out_channel = struct
  type channel = out_channel
  let char = output_char
  let substring = output
end

module BufferOutput : OUTPUT with type channel = Buffer.t = struct
  type channel = Buffer.t
  let char = Buffer.add_char
  let substring = Buffer.add_substring
end
```

Notez que de cette façon les concepteurs de Menhir peuvent écrire tout leur code d'affichage (dans le fichier `pprint.ml`) comme un foncteur qui est défini comme

```
(* The renderer is parameterized over an implementation of output channels. *)

module Renderer (Output : OUTPUT) = struct
...
```

et l'instancier simplement en écrivant

```
(* Instantiating the renderers for the two kinds of output channels. *)

module Channel =
  Renderer(ChannelOutput)

module Buffer =
  Renderer(BufferOutput)
```

Exercice : identifier parmi vos enseignants un des auteurs de Menhir.

4 Banques

Exercice 7

1. Donnez deux signatures `Client` et `Banque` de module qui représentent une banque. L'une est destinée au client et comprendra les fonctions `depot` et `retrait` toutes deux de type `t -> monnaie -> monnaie`. Les types `t` et `monnaie` seront abstraits. L'autre signature est destinée au banquier et aura en plus de celle du client la possibilité de créer une Banque.
2. Donnez une implémentation du module `Banque` de type :

```
module Banque : functor (M : MONNAIE) ->
  sig
    type t
    type monnaie = M.t
    val creer : unit -> t
    val depot : t -> monnaie -> monnaie
    val retrait : t -> monnaie -> monnaie
  end
```

   Vous donnerez un type concret pour `t`.
3. Créez une banque `Toto` tenant ses comptes en euros.
4. Donnez une implémentation de la signature `Client` qui soit compatible avec la banque `Toto` précédemment crée i.e. on peut faire :

```
let mon_ccp = Toto.creer ();;
Toto.depot mon_ccp (euro 100.0);;
Client.depot mon_ccp (euro 100.0);;
```

5. Créez une banque gérant les comptes en dollars et vérifier qu'il n'est pas possible d'ajouter des euros.

Correction :

```
module type CLIENT =            (* vue donnée au client *)
  sig
    type t
    type monnaie
    val depot : t -> monnaie -> monnaie
    val retrait : t -> monnaie -> monnaie
  end
;;
```

```
module type BANQUE =            (* vue du banquier *)
  sig
    type t
    type monnaie
    val creer : unit -> t
    val depot : t -> monnaie -> monnaie
    val retrait : t -> monnaie -> monnaie
  end
;;

(* on pourrait aussi ecrire
module type BANQUE =
  sig
    include CLIENT
    val creer: unit -> t
  end
 *)

module Banque (M : MONNAIE) :
    BANQUE with type monnaie = M.t =
  struct
    type monnaie = M.t and t = monnaie ref
    let zero = M.prod 0.0 M.un and neg = M.prod (-1.0)
    let creer () =  ref zero
    let depot c x =
      if x > zero then c := M.plus !c x; !c
    let retrait c x =
      if !c > x then (c := M.plus !c (neg x); !c) else zero
  end;;

module Toto = Banque (Euro);;

module Client = (Toto : CLIENT
     with type monnaie = Toto.monnaie
     with type t = Toto.t)
;;

let mon_ccp = Toto.creer ();;
Toto.depot mon_ccp (euro 100.0);;
Client.depot mon_ccp (euro 100.0);;

module Citybank = Banque (Dollar);;
let mon_compte_aux_US = Citybank.creer ();;


(* Verification de l'incompatibilite *)
(* Citybank.depot mon_ccp;;                  *)
(* Citybank.depot mon_compte_aux_US (euro 100.0);;  *)
```

5 Plus de foncteurs

Exercice 8 [Utilisation de foncteurs] Récupérez le code à l'adresse `http://www.dicosmo.org/CourseNotes/pfav/tp/tp_code1.ml`

1. Grâce au foncteur `Set`, créez un un module représentant les ensembles de chaînes de caractères.
2. Essayez de créer un ensemble contenant les éléments "abc", "def", "ghi". Quel est le problème ?
3. Après avoir résolu le problème, créez l'ensemble demandé à la question précédente et vérifiez le résultat de la fonction mem sur différentes valeurs.
4. Réécrivez le module `Set` en utilisant la notation simplifiée pour les foncteurs.

Correction :

- `module StringSet = Set (String)`
- Les types sont différents. Il faut modifier la définition avec une contrainte de type qui permet de garder l'égalité entre le type `content` et le type `t` :

```
module Set (T:Comparable) : SET with type content = T.t =
   struct
     ...
```

6 Foncteurs de la bibliothéque standard

Exercice 9 [Les foncteurs `Set` et `Hashtbl`]

- Ecrivez un module Tree représentant des arbres binaires d'entiers. Créez ensuite une table de hachage avec pour éléments ces arbres en utilisant le foncteur Hashtbl.Make. Pour la fonction de hachage, vous pourrez utiliser la fonction polymorphe du module Hash.
- Créez également un module contenant un ensemble d'arbres binaires ordonnées par leurs tailles grâce au foncteur Set.Make de la librairie standard (attention aux problème de visibilitè si vous êtes dans le même environnement que l'exercice précédent !)
- Créez maintenant une table de hachage avec pour éléments des ensembles d'arbres (il peut être utile de recourir à l'instruction include).
- Testez votre code en créant des ensembles d'arbres binaires.

Correction :

```
(* Un module definissant des arbres binaires avec des fonctions *)
(* de comparaison et de hachage                                 *)

module Tree = struct
  type tree = Leaf | Node of int * tree * tree

  type t = tree

  let rec equal t1 t2 = match (t1, t2) with
    | Leaf, Leaf -> true
    | Node(x, lt1, rt1), Node (y, lt2, rt2) ->
        x==y && equal lt1 rt1 && equal rt1 rt2
    | _ -> false

  let hash tree = Hashtbl.hash tree

  let rec size t = match t with
    | Leaf -> 0
    | Node(x, lt, rt) -> 1 + size lt + size rt
```

```
  let compare t1 t2 = size t1 - size t2
end

(* On a tout le necessaire pour creer une table de hachage *)

module HTree = Hashtbl.Make (Tree)

(* et aussi des ensembles *)

module Ordered_Tree_Set = Set.Make(Tree)

(* pour creer une table de hachage, il nous manque une fonction
   de hachage dans la signature de Ordered_Tree_Set...
   rajoutons la concisement en utilisant l'inclusion *)

module TreeSet =
struct
  include Set.Make (Ordered_Tree_Set)
  let hash set = Hashtbl.hash set
end

(* maintenant on peut appliker Hashtbl.Make ! *)

module HTreeSet = Hashtbl.Make  (TreeSet)
```