info401 : Programmation fonctionnelle
TD 6 : modules et types abstraits, références

Pierre Hyvernat
Laboratoire de mathématiques de l'université de Savoie
bâtiment Chablais, bureau 22, poste : 94 22
email : Pierre.Hyvernat@univ-savoie.fr
www : http://www.lama.univ-savoie.fr/~hyvernat/
wiki : http://www.lama.univ-savoie.fr/wiki

Exercice 1 : les ensembles

Question 1. La notion d'ensemble fini est couramment utilisée en informatique. Donnez une signature possible (appelée `E`) pour un module `Ensemble` où le type des ensembles est abstrait.

Question 2. On peut assez facilement implémenter les ensembles finis en utilisant des listes triées. Pour ceci, il est pratique d'avoir une notion de *type ordonné* :

```
module type OrderedType = sig
  type t
  val compare : t -> t -> int
end
```

Remarque : tous les types de Caml peuvent devenir des *types ordonnés* en utilisant la fonction polymorphe `compare` de Caml. Par contre, cette fonction ne fonctionne que sur les types de données (et pas sur les types fonctionnels) ; et on peut avoir envie d'utiliser une autre fonction de comparaison...

Écrivez un foncteur qui commence de la manière suivante :

```
module FaireEnsemble (T:OrderedType) : E with type elt=T.t =
struct
...
```

Question 3. Si le type `t` est un type de données (et a donc une fonction de comparaison qui fonctionne), on peut écrire une fonction `supprime_doublons : t list -> t list` qui utilise un ensemble pour stocker les doublons : la fonction parcourt la liste et met les éléments qu'elle trouve dans un ensemble. Elle conserve les éléments qui apparaissent pour la première fois, mais pas ceux qui ont déjà étés mis dans l'ensemble en question.

Écrivez, cette fonction et expliquez ce que vous faites.

Question 4. Caml possède, dans la librairie `Set`, une implémentation des ensembles finis (la signature correspondante est `S`) et un foncteur `Make` :

```
module Make (Ord : OrderedType) : S with type elt = Ord.t
```

Cette implémentation est basée sur les arbres binaires de recherche bien équilibrés. Elle est donc plus efficace que notre version utilisant les listes.

Expliquez comment modifier la fonction `supprime_doublons` pour qu'elle utilise l'implémentation des ensembles de Caml.

Exercice 2 : références

Question 1. Imaginez que vous avez un gros programme comportant de nombreuses fonctions. Une de ces fonctions, la fonction `f`, vous intéresse particulièrement, et vous aimeriez savoir combien de fois vous l'appelez lors d'une exécution de votre programme.

Expliquer comment vous procédez. (Sans utiliser de références.)

Question 2. Même question, mais en utilisant une référence. Est-ce que la transparence référentielle est conservée ?

Quelle solution préférez-vous ?

Question 3. Toujours à propos de la fameuse fonction `f`, vous aimeriez optimiser le calcul des valeurs. Lors du calcul de `f x`, vous regardez si vous avez déjà calculé la valeur correspondante.

- Si oui, vous renvoyez la valeur déjà calculée
- si non, vous calculez `f x` normalement, et conservez la valeur quelque part.

(Cette technique s'appelle "mémoization". Le deuxième calcul de `f x` est en général beaucoup plus rapide que le premier...)

Expliquez comment vous procéderiez, sans utiliser de référence.

Que pensez-vous de la complexité de `memoization f` par rapport à celle de `f` ?

Question 4. Même question, mais en utilisant des références. Que pensez-vous de la complexité de `memoization f` par rapport à celle de `f` ?

Est-ce que la transparence référentielle est conservée ?

Quelle solution préférez-vous ?

Question 5. En utilisant les remarques des questions précédentes, écrivez une fonction `mémoization : ('a->'b) -> ('a->'b)` qui transforme une fonction (presque) quelconque en une version mémoizée.

Est-ce que la transparence référentielle est conservée ?

Question 6. Expliquez comment vous pourriez améliorer encore un peu la complexité de votre fonction...