Université de Bordeaux
Année 2019-2020

Mastère 2
T.A.P

TD n°6 - Programmation paresseuse

Exercice 1: Introduction à la congélation

Nous étudions ici quelques aspects de la *congélation*. Une manière d'implémenter cette technique consiste à préfixer les expressions par un niveau de fonctionnelle `fun () → ...` de manière à ne pas évaluer directement l'expression. Celle-ci sera évaluée lors de l'opération de *dégel*, qui consiste à forcer l'évaluation de ce niveau de fonctionnelle. La congélation permet ainsi de maîtriser l'exécution des calculs.

Considérons le type suivant pour les valeurs :

```
type 'a frozen_flow =
| End of 'a
| Step of (unit → 'a frozen_flow);;
```

Une valeur est donc soit calculée, soit congelée et en attente de décongélation.

1. Écrire la fonction `thaw :'a frozen flow → 'a frozen flow` qui permet de décongeler une étape de calcul (si elle est congelée).

Le code suivant permet de calculer de manière réursive terminale le ppcm de deux nombres entiers positifs :

```
let ppcm x y =
  let rec ppcm_rec x y mul =
    if (y > x) then (ppcm_rec y x mul) else (* Ensures x >= y *)
    if (x = 0) then 0 else                  (* Ensures both are positive *)
      let r = (x mod y) in
        if (r = 0) then (mul/y) else ppcm_rec y r mul
  in ppcm_rec x y (x*y);;
```

2. Réécrire cette fonction en utilisant le type `frozen_flow` pour effectuer ce calcul en congelant *chaque* appel récursif.

   Indice : son type doit devenir `int → int → int frozen_flow`.

Enrichissons le type `frozen_flow` pour permettre de manipuler en même temps que le flot d'exécution des données présentes dans l'environnement sous fome d'une liste d'associations (associant un nom de variable et une valeur, dans le cas présent quelque chose comme `[("f",6);("n",4)]`)

```
type ('key,'data) frozen_data_flow =
| End of 'data
| Step of (unit → (('key*'data) list) * ('key,'data) frozen_data_flow);;
```

3. Réécrire la fonction précédente en utilisant ce nouveau type, de façon à obtenir une forme primitive de débogueur.

Exercice 2: Utilisation de la bibliothèque Lazy

La bibliothèque standard d'OCaml dispose d'un module `Lazy` afin de faciliter l'utilisation de types paresseux (`http://caml.inria.fr/pub/docs/manual-ocaml/libref/Lazy.html`). Concrètement, on utilise le mot-clé `lazy`, qui prend en argument un élément de type `'a`, et qui renvoie un élément de type `'a lazy_t` (sans évaluer l'élément) [1]. L'évaluation forcée de l'élément est obtenue par la fonction `Lazy.force`.

```
let f = lazy ( 1/0 );; (* int lazy_t = <lazy> *)
Lazy.force f;;
```

1. Congeler un entier, une liste, un tableau, puis les décongeler.
2. Congeler une fonction réalisant un effet de bord (par un affichage par exemple), puis la décongeler. A quel moment l'effet de bord se produit-il ?
3. Que se passe-t'il lorsque l'on essaie de décongeler plusieurs fois de suite les expressions précédentes ?

Exercice 3: Flots de données

Les *flots de données* sont des types de données représentant les données sous forme de liste. Néanmoins, contrairement aux listes, les données ne sont pas évaluées lors de la définition de la liste, mais au moment où ces données sont utilisées par une fonction. Pour représenter les flots (ou *stream* en anglais), on peut utiliser le type inductif suivant :

```
type 'a stm =
| StmEmpty
| StmCons of ('a * 'a stm) lazy_t
```

De cette manière, un flot est soit vide, soit une valeur paresseuse dont l'évaluation renvoie une paire contenant le premier élément et la suite de la liste. Pour vérifier votre code, la fonction suivante permet de compter le nombre d'éléments déjà évalués dans un flot :

1. Remarquer que `lazy` ne peut pas être une fonction du langage, parce qu'une fonction évalue forcément ses arguments, chose que l'on désire absolument éviter pour utiliser la paresse. Ce comportement est similaire au `defmacro` du Lisp.

```
let rec length_evaluated stm = match stm with
  | StmEmpty   → 0
  | StmCons(x) → if not(Lazy.is_val x)
                   then 0
                   else let (_,v) = Lazy.force x in
                        1 + length_evaluated v;;
```

1. Écrire des instances du type `int stm`, contenant au moins une fois un élément dont le calcul lève une exception, comme `1/0`. Quel est le principal inconvénient levé par cette définition des flots ?
2. Implémenter les accesseurs `stm_head` et `stm_tail`, ainsi que la fonction qui construit la liste des n premiers éléments d'un flot.
3. Écrire une fonction `list_to_stream` permettant de transformer une liste en un flot de données.
4. Écrire une fonction `fun_to_stream_bounded` qui, étant donnée une fonction f, permet de construire le flot contenant $[f(1), f(2), \ldots, f(n)]$.

▷ Résumons les idées des exercices précédents. Pour construire des flots, deux possibilités s'offrent à nous :

(a) construire une liste et transformer la liste en flot (ce qui est contraire l'idée de ne pas évaluer les éléments du flot) ;

(b) construire directement le flot en encapsulant dans le flot la fonction qui construit la fin du flot (comme lorsque l'on a congelé le calcul du PPCM).

Pour la deuxième possibilité, supposons disposer d'une fonction `f : 'a → 'a` qui calcule le $n+1$-ème élément du flot à partir du n-ème élément.

5. Écrire une version `fun_to_stream_unbounded` de la fonction précédente prenant en argument la fonction `f`.

   Remarquer qu'il est possible de réécrire la fonction *bounded* à partir de la fonction *unbounded* en lui passant en paramètre une fonction du style :

   `fun (n,_) → (n+1, List.nth l (n+1))` où `l` est la liste de départ

   Quelle différence fondamentale existe t'il entre la version *bounded* et la version *unbounded* ? Qu'est-ce qui, dans la programmation paresseuse, permet de décrire un tel comportement non borné ?
6. Écrire une fonction `stm_map` qui, étant donné un flot et une fonction f, renvoie le flot où tous les éléments du flot ont été transformés par f.
7. Écrire une fonction `stm_compose` qui, étant donné deux flots et une fonction de composition (des éléments de ces deux flots), renvoie le flot des éléments composés.
8. Écrire une fonction `stm_concat` qui, étant donné deux flots, renvoie le flot qui les concatène.

Exercice 4: Arbres paresseux

Supposons vouloir adapter la notion de programmation paresseuse à des structures de données non séquentielles, comme par exemple des arbres. Pour cela, l'idée va être de construire des arbres dont les noeuds contiennent des valeurs `lazy`, et dont les fils sont des *flots de noeuds*. Pour commencer, considérons les arbres non paresseux suivants :

```
type 'a tree = TmEmpty  | TmCons of ('a * 'a tree list)

(* Builds the list of integers [a,a+1,..,b-1] *)
let rec list_build a b = if (a>=b) then [] else a::(list_build (a+1) b);;

(* Builds a tree with a generating function f, depth n and top node start *)
(* tree_build : ('a → 'a list) → int → 'a → 'a tree *)
let rec tree_build f n start =
    let nodel = f start in
    let treel = if (n<=0) then [] else
                  List.map (tree_build f (n-1)) nodel in
    TmCons (start, treel);;

(* Builds a tree of integers of depth d and branching k *)
let tree_interv k d =
  tree_build (fun x → list_build (k*x-k+2) (k*x+2)) d 1;;
```

La fonction `tree_build` prend en paramètre une fonction `f` qui, étant donnée une valeur de type `'a`, renvoie la liste des valeurs "filles" de type `'a list`.

1. Construire à partir de la fonction `tree_interv` donnée dans le code un arbre d'entiers. Comment fonctionne l'appel récursif dans la fonction `tree_build` ?

Appliquons maintenant notre construction à des arbres dans lesquels les listes sont remplacées par des flots.

```
type 'a stm  = StmEmpty | StmCons of ('a * 'a stm) lazy_t
type 'a lazytree = LTEmpty | LTCons of ('a lazy_t) * (('a lazytree) stm)
```

2. Écrire une fonction `lazytree_build` équivalent de la fonction `tree_build` pour les arbres paresseux.