Université Montpellier II — Master d'Informatique
UMINM 121 Evaluation des Langages Applicatifs — 2 heures

R. Ducournau – M. Lafourcade

février 2006

Documents autorisés : notes de cours, polys, pas de livre. Notation globale sur 26.

Les flots et l'évaluation retardée

On se propose d'étudier une généralisation des listes plates connue sous le nom de *flots*. Un flot est une liste plate dont l'évaluation de la queue (CDR) est retardée. Les flots permettent ainsi de manipuler des listes virtuellement infinies.

Les flots vont être d'abord définis par leur implémentation dans le méta-évaluateur du cours (première génération) et les questions porteront sur ce que font les programmes sur les flots.

Dans l'analyse par cas (`cond`) de la fonction `meval`, on rajoute les clauses suivantes pour les deux formes spéciales `delay` et `force` :

```
(defun meval (expr env)
    (cond ....
          ((eq 'delay (car expr))
           (meval #'(lambda () ,(cadr expr)) env))
          ((eq 'force (car expr))
           (meval-closure (meval (cadr expr) env) ()))
          ....))
```

Le principe est le suivant :

1. `delay` fabrique une fermeture sans paramètre sur son argument non évalué, et la retourne : c'est donc une *évaluation retardée* (ou une promesse d'évaluation) ;
2. `force` prend en argument une fermeture sans paramètre et l'applique : c'est l'évaluation de l'expression retardée par `delay` ;

Une fois ces formes spéciales définies, il est possible de définir des fonctions de manipulations de flots, en particulier `vide-f`, `tete-f` et `queue-f` qui sont les analogues de `null`, `car` et `cdr` pour les flots :

```
(defun vide-f (flot) (null flot))
(defun tete-f (flot) (car flot))
(defun queue-f (flot) (force (cdr flot)))
```

NB Il s'agit bien entendu de *méta-définitions* : le code de ces 3 fonctions doit forcément être méta-évalué, puisque les flots ne sont pas connus par `eval`.

Question 1

Rappeler comment les fermetures sont traitées dans le méta-évaluateur. Traiter les cas de `function` et `apply`. NB Dans le code de méta-évaluation de `force`, `meval-closure` correspond justement à l'application d'une fermeture.

Question 2

Cette mécanique de base une fois définie, il est possible de définir des flots. Par exemple, la fonction suivante définit le flots des entiers à partir de son argument `n` :

```
(defun enumerer-f (n) (cons n (delay (enumerer-f (1+ n)))))
```

NB encore une fois, la définition précédente et les expressions suivantes doivent être évaluées par `meval`, pas par `eval`. Quelle est la valeur retournée par les expressions (Donnez l'expression Lisp ou l'arbre de doublets, au choix)

1. `(enumerer-f 6)` ?
2. `(queue-f (queue-f (enumerer-f 6)))` ?
3. `(tete-f (queue-f (queue-f (queue-f (enumerer-f 6)))))` ?

Question 3

Ecrire les fonctions suivantes sur les flots :

1. `(print-flot flot n)` imprime les `n` premiers éléments d'un flot et retourne `:end` si lorsque le flot s'arrête avant la fin, et `:overflow` lorsque le flot n'est pas épuisé avec les `n` premiers éléments.
2. `(flot2list flot n)` fait la liste des `n` premiers éléments d'un flot ;
3. `(list2flot list)` retourne un flot qui énumère les éléments de la liste arguments.
4. `(merge-flot flot1 flot2)` retourne un flot qui mélange les 2 flots arguments, de telle sorte que les 2 flots soient consommés à la même vitesse : ce n'est pas une concaténation mais un mélange.

Les flots permettent une programmation efficace et élégante par passage de flots : par exemple, l'expression

```
(somme-f (carre-f (filtre-impairs-f (generer-feuilles-f arbre))))
```

prend un `arbre` binaire de cellules, génère le flot de ses feuilles, le passe à une fonction qui retourne le flot filtré des valeurs impaires, pour le passer à une fonction qui retourne le flot de ces valeurs au carré, pour le passer à un flot qui en fait la somme.

Ainsi, en traduisant les flots en listes par souci de simplicité, l'arbre `((1 . 2) . (3 . 4))` donnera successivement `(1 2 3 4)`, `(1 3)`, `(1 9)` puis finalement retournera `10`.

Question 4

Ecrire les fonctions `somme-f`, `carre-f`, `filtre-impairs-f` et `generer-feuilles-f`, en vous servant au besoin de fonctions de la question précédente.

Question 5

On souhaite remplacer l'appel explicite à `delay` dans `(cons x (delay y))` par une "fonction" `cons-stream`, de paramètre `x` et `y`.

En donner la définition dans `meval` comme forme spéciale ou sous forme de macro. Serait-il possible d'en faire une vraie fonction ? Pourquoi ?

Question 6

L'implémentation des flots proposée ici a l'inconvénient de ne pas mémoriser le résultat de l'évaluation retardée : dans le code

```
(let ((flot (enumerer-f 6)))
    (queue-f flot)
    (queue-f flot))
```

le forçage par l'appel de `queue-f` est effectué 2 fois, la deuxième inutilement.

1. Proposer une solution basée sur la chirurgie, qui modifie physiquement le flot. Comment faut-il modifier la méta-évaluation de `force` ?
2. Une solution alternative consiste à modifier la méta-évaluation de `delay` en capturant des variables supplémentaires qui mémorisent le résultat du forçage précédent. Comment ?
3. dans l'un ou l'autre cas, que retourne le code suivant :

```
(let ((flot (enumerer-f 6)))
    (queue-f flot)
    flot)
```

En savoir plus

Ce problème de flots est longuement décrit dans le livre d'Abelson et Sussman, "Structure et interprétation des programmes informatiques" (chapitre 3.4).