Université Montpellier II — Master d'Informatique
HMIN 104 Compilation et interprétation — 2 heures

R. Ducournau – M. Lafourcade

Mars 2016

Documents autorisés : notes de cours, polys, pas de livre.
Notation globale sur 25.

Le point de départ de ce sujet est le langage intermédiaire (Chapitre 5 du polycopié "Compilation et Interprétation des Langages"). On suppose la transformation effectuée (par la fonction `lisp2li` du cours) et on s'intéresse ici à des transformations, écrites en LISP, d'expressions du langage intermédiaire.

On se restreindra à la partie du langage dont la syntaxe des *expressions évaluables* (`<expr-eval-li>`) est la suivante :

<expr-eval-li>	:=	« (:**const** . » <expr> «) » |
		« (:**var** . » <int> «) » |
		« (:**if** » <expr-eval-li> <expr-eval-li> « . » <expr-eval-li> «) » |
		« (:**progn** » <expr-eval-li> <expr-eval-li>+ «) » |
		« (:**set-var** » <int> « . » <expr-eval-li> «) » |
		« (:**mcall** » <symbol> <expr-eval-li>* «) » |
		« (:**call** » <symbol> <expr-eval-li>* «) » |
		« (:**unknown** » <expr-eval-lisp> « . » <env> «) »

1 Appels terminaux

Soit l'exemple LISP d'une fonction `foo` qui appelle une fonction `bar` :
`(defun foo (...) ... (bar ...) ...)`
où les ... représentent n'importe quelles expressions LISP qui font de l'appel de `bar` une sous-expression évaluable du corps de `foo`.

Un appel de fonction, par exemple l'appel de `bar` dans `foo`, est terminal si la fonction appelante (`foo`) retourne le résultat renvoyé par la fonction appelée (`bar`) sans effectuer le moindre traitement (effet de bord ou calcul) entre le retour de l'appelée et le retour de l'appelant.

C'est une généralisation de la notion de récursion terminale : dans ce dernier cas, on a `bar` =`foo`. Tout appel terminal n'est pas récursif. Inversement, une fonction récursive enveloppée contient certainement un appel terminal.

1.1 Mise en jambes (sur 5)

On définit d'abord un exemple :

Question 1

Soit la fonction récursive sur les entiers suivante :

$$mc(x) = \begin{cases} x - 10 \text{ si } x > 100 \\ mc(mc(x + 11)) \text{ sinon} \end{cases} \quad (1)$$

1. donner d'abord la définition en LISP de la fonction `mc` ;
2. donner alors la valeur fonctionnelle de `mc` telle qu'elle aura été produite par `lisp2li` dans le langage intermédiaire (et telle qu'elle sera retournée par `(get-defun 'mc)`) ;
3. donner ensuite le résultat de l'application de `lisp2li` sur l'expression `(defun mc (x) ...)` qui définit `mc` ;

4. donner enfin la valeur fonctionnelle de `mc` une fois que la fonction aura été utilisée, ce qui provoquera l'expansion des `:unknown`.
   ATTENTION : dans la suite, on considérera que la valeur fonctionnelle de `mc` est dans cet état là.

1.2 Détection des appels terminaux (sur 10)

Question 2

On généralise avec la notion de sous-expression terminale. Soit une expression e d'un langage. Une sous-expression f de e est terminale dans e si l'évaluation de e se termine par l'évaluation de f en retournant sa valeur, sans qu'aucun traitement ne soit effectué entre le retour de e et celui de f.

1. Montrer que la notion de sous-expression terminale est transitive : si g est terminale dans f et que f est terminale dans e, alors g est terminale dans e.
   En déduire qu'un appel (`:call` ou `:mcall`) est terminal si c'est une sous-expression terminale de toutes les expressions qui le contiennent.
2. Dans chacun des cas suivants d'expressions évaluables, indiquer pour chaque sous-expression si elle est terminale ou pas.

« (**:if** » <expr-eval-li-1> <expr-eval-li-2> « **.** » <expr-eval-li-3> «) » |
« (**:progn** » <expr-eval-li-1> <expr-eval-li-2> ... <expr-eval-li-n> «) » |
« (**:set-var** » <int> « **.** » <expr-eval-li> «) » |
« (**:mcall** » <symbol> <expr-eval-li-1> <expr-eval-li-2> ... <expr-eval-li-n>«) » |
« (**:call** » <symbol> <expr-eval-li-1> <expr-eval-li-2> ... <expr-eval-li-n> «) » |
« (**:unknown** » <expr-lisp> « **.** » <env> «) »

---

3. Est-il possible qu'il y ait 2 sous-expressions terminales dans la même expression ? Expliquer pourquoi.
4. dans quel cas le corps d'une fonction ne contient-il aucun appel terminal ?

Question 3

On étend la grammaire du langage intermédiaire avec le mot-clé `:mcallt` pour les appels terminaux de fonctions méta-définies et `:callt` pour les appels terminaux de fonctions prédéfinies.

Modifier la définition de la fonction `mc` dans le langage intermédiaire de façon à utiliser le mot-clé approprié pour chaque appel.

Question 4

Spécifier et écrire la fonction LISP `(marque-terminal-li fun)` qui prend en paramètre (`fun`) le nom d'une fonction définie dans le langage intermédiaire et transforme sa valeur fonctionnelle en remplaçant, pour les appels terminaux, `:call` et `:mcall` par `:callt` et `:mcallt`.

1.3 Appels terminaux et machine à registres (sur 10)

(On fait ici référence à la machine virtuelle à registres - Chapitre 7 du polycopié "Compilation et Interprétation des Langages")

Question 5

Donner la définition de la fonction `mc` dans le code assembleur de la machine virtuelle à registres (tel qu'il serait produit par `li2vm`).

Question 6

Expliquer comment modifier le code VM de `mc` pour optimiser les appels récursifs terminaux de façon à ce que l'exécution de `mc` ne consomme pas de pile dans ces cas-là. (NB on ne cherche pas à optimiser les appels terminaux qui ne sont pas récursifs.)

1.4 Par curiosité

Quelle valeur est retournée par la fonction `mc` ?