Université Montpellier II — Master d'Informatique
FMIN 106 Compilation — 2 heures

R. Ducournau – M. Lafourcade

Juin 2010

Documents autorisés : notes de cours, polys, pas de livre.
Notation globale sur 23.

Le point de départ de ce sujet est le langage intermédiaire du méta-évaluateur "threadé" (Chapitre 5 du polycopié "Compilation et Interprétation des Langages"). On suppose la transformation effectuée (par la fonction appelée `lisp2li` dans le cours) et on s'intéresse uniquement à des analyses et 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> «) »

Il y a donc dans l'ordre des mots-clés pour : les constantes et les variables, une forme conditionnelle, la séquence, l'affectation de variable, l'appel de fonction méta-définie, et de fonction prédéfinie, et enfin le cas des expressions encore inconnues au moment de la transformation.

Par rapport à Lisp, le langage est grandement simplifié : il n'y a plus de `let` (la transformation est censée les avoir fait disparaître, suivant le principe décrit dans le poly) et on ne s'intéresse pas ici à des constructions de plus haut niveau comme des fermetures, les fonctions locales ou les mots-clés `&rest` ou `&optional`. Il n'y a bien sûr plus de macros—sauf dans la forme :`unknown` si la macro a été définie après la transformation du code considéré.

La définition d'une fonction méta-définie, c'est-à-dire sa "valeur fonctionnelle", est constituée d'une paire contenant l'expression unique qui forme le corps de la fonction et un entier qui indique la taille de l'environnement à créer (ou la réservation de pile à faire) lors de l'appel.

On suppose de plus qu'il existe deux fonctions Lisp `get-defun` et `set-defun` qui permettent d'accéder à un symbole pour récupérer ou affecter sa "valeur fonctionnelle". Ainsi `(defun foo ...)` se traduit par `(set-defun 'foo <valeur fonctionnelle>)`. En contrepartie, on ne suppose l'existence d'aucun environnement fonctionnel.

1 Exemple de langage intermédiaire (sur 8)

Question 1

La fonction d'Ackerman a la définition suivante, où m et n sont des entiers positifs ou nuls :

$$\texttt{acker}(m,n) = \begin{cases} n+1 & \text{si } m = 0, \\ \texttt{acker}(m-1,1) & \text{si } m > 0 \text{ et } n = 0, \\ \texttt{acker}(m-1,\texttt{acker}(m,n-1)) & \text{si } m > 0 \text{ et } n > 0 \end{cases}$$

1. Ecrire d'abord en Lisp la définition de la fonction `acker`, dans sa forme récursive directe ; on supposera que la valeur des paramètres m et n est correcte (ce sont bien des entiers positifs ou nuls) et on simplifiera la définition au maximum, en particulier pour éviter les constructions qui ne se traduisent pas directement dans le langage intermédiaire ;
2. puis en donner la valeur fonctionnelle dans le langage intermédiaire, telle qu'elle serait produite par `lisp2li` ;

3. écrire dans le langage intermédiaire l'expression que doit retourner `(lisp2li '(defun acker (m n) ...))` pour que son évaluation associe la valeur fonctionnelle au symbole `acker` ;
4. donner enfin la valeur fonctionnelle de `acker` une fois que le corps de la fonction aura été évalué.

2 Parcours d'arbres et analyse par cas (sur 6)

La manipulation de langages formels est une histoire de parcours d'arbres et d'analyse par cas.

Question 2

Le parcours doit être *correct* (il ne traite que des expressions évaluables du langage intermédiaire) et *complet* (il les traite toutes). Il doit donc correspondre très exactement à la syntaxe de `<expr-eval-li>` ci-dessus.

1. Définir la fonction Lisp `parcours-arbre-li` (`pal` pour faire court) qui prend en paramètre une expression évaluable du langage intermédiaire et parcourt récursivement toutes ses sous-expressions évaluables (elles-mêmes du langage intermédiaire).

   Cette fonction ne fait rien mais dans les questions suivantes on appliquera le même schéma pour faire différents traitements sur une expression.
2. Pour donner de la consistance à cette fonctionnalité, définir la fonction Lisp `parcours-arbre-li-example` (`pale` pour faire court) qui
   - prend en entrée une expression évaluable du langage intermédiaire,
   - parcourt toutes ses sous-expressions évaluables,
   - en imprimant le mot-clé de l'expression et sa profondeur dans l'arbre,
   - et retourne le nombre d'expressions parcourues.

3 Sous-expressions terminales (sur 10)

Soit une expression e du langage intermédiaire. 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.

Question 3

1. Montrer que cette notion est transitice : si g est terminale dans f et que f est terminale dans e, alors g est terminale dans e.
2. Dans chacun des cas suivants, quelles sont les sous-expressions terminales ?

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

Un appel terminal est une sous-expression terminale de forme `:call` ou `:mcall` dans le corps d'une fonction. C'est une généralisation de la notion de récursion terminale. On étend la grammaire du langage avec le mot-clé `:mcallt` pour les appels terminaux de fonctions méta-définies et `:callt` pour les appels terminaux de fonctions prédéfinies.

Question 4

Modifier la définition de la fonction `acker` (question 1) dans le langage intermédiaire de façon à utiliser le bon mot-clé pour les appels terminaux.

Question 5

Spécifier et écrire la fonction Lisp `marque-terminal` qui à partir d'une expression en langage intermédiaire qui représente le code d'une fonction, transforme l'expression pour tenir compte des appels terminaux en se servant de `:callt` et `:mcallt`.

Traiter spécifiquement les différents cas de récursion de `marque-terminal`, comme dans la Question 3.