Examen Compilation

Université Paris Diderot, Master Ingénierie Informatique (M1).

Deuxième session 2011-2012. Durée 2h. L'utilisation de notes de cours est autorisée, l'utilisation de tout autre document ou dispositif électronique est interdite.

Exercice 1 *On se place dans le cadre du problème de la sélection d'instructions. On prend comme signature :* $\Sigma = \{a^0, g^2\}$. *On suppose les tuiles (filtres) et coûts suivants :*

Tuile	*Coût*
$f_1 = a$	1
$f_2 = g(a, a)$	2
$f_3 = g(x, y)$	3
$f_4 = g(g(x, y), g(w, z))$	4

(1) *Montrez qu'il y a* toujours une solution *au problème de recouvrement. Quels sont les* ensembles minimaux *de tuiles parmi celles considérées qui ont cette propriété ?*

(2) *Considérez l'arbre* $t_0 = g(g(a, a), g(a, a))$. *Quels sont les* recouvrements possibles *et quels sont les* recouvrements de coût minimal *?*

Pour trouver une solution optimale, on se propose d'utiliser la méthode de programmation dynamique suivante.

- *On énumère les noeuds de l'arbre des feuilles vers la racine (*bottom-up*).*
- *Quand on est au noeud n on a déjà déterminé le coût minimum de tous les* descendants *de n.*
- *On énumère toutes les* tuiles qui sont compatibles *avec le noeud n (le sous-arbre de racine n matche le filtre). Pour chaque tuile t de coût c il y aura un certain nombre de sous-arbres de racine* $n_1, \ldots, n_k$ *(*$k \geq 0$*) qui ne sont pas couverts par la tuile et dont le coût minimum est* $c_1, \ldots, c_k$.
- *On dérive que* $c + \Sigma_{i=1,\ldots,k} c_i$ *est une* borne sup au coût *du sous-arbre n. Le* coût *de n est le minimum des bornes sup.*

(3) Appliquez *la méthode de programmation dynamique à l'exemple (arbre* t_0*).*

(4) Expliquez *pourquoi cet algorithme trouve une solution optimale (dans l'exemple et en général).*

Exercice 2 *Dans le cours nous avons discuté une extension du langage Imp avec fonctions, disons ImpF ; cet exercice continue la discussion vers une mise en oeuvre.*

- *La catégorie syntaxique des* **commandes** *est étendue comme suit :*

$$S ::= \mathsf{skip} \mid id := e \mid S; S \mid \mathsf{if}\ b\ \mathsf{then}\ S\ \mathsf{else}\ S \mid \\ \mathit{while}\ b\ \mathsf{do}\ S \mid id := f(e, \ldots, e) \mid \mathsf{return}(e)$$

- *Une mémoire* locale *est une fonction partielle $ls \in [id \mapsto \mathbf{Z}]$. La mémoire m est maintenant représentée par un couple (ls, s) et les fonctions de lecture et de mise-à-jour sont adaptées en conséquence.*
- *La catégorie syntaxique des* **continuations** *est étendue comme suit :*

$$K ::= \mathsf{halt} \mid \mathsf{returnto}(f, id, K, ls) \mid S \cdot K$$

- *La sémantique à grands pas des expressions et des conditions booléennes et à petits pas des commandes repose sur des jugements de la forme :*

$$(e, m) \Downarrow v \qquad (b, m) \Downarrow v \qquad (f, S, K, m) \to (f', S', K', m')$$

- *Un programme est une suite de déclarations de fonctions dont la dernière est le* 'main()'. *Une déclaration de fonction a la forme :*

$$f(x_1, \ldots, x_n) = \mathsf{var}\ x_{n+1}, \ldots, x_m; S$$

où l'on peut supposer que les variables $x_1, \ldots, x_m$, sont toutes différentes et que les variables locales *$x_{n+1}, \ldots, x_m$ sont initialisées avec la valeur 0. Le passage des paramètres est* par valeur.

On utilise les notations $m(x)$ pour lire le contenu de la variable x dans la mémoire m et $m[v/x]$ pour mettre à jour le contenu de la variable x dans la mémoire m. Formellement on a :

$$(ls, s)(x) = \begin{cases} ls(x) & si\ x \in dom(ls) \\ s(x) & autrement \end{cases} \qquad (ls, s)[v/x] = \begin{cases} (ls[v/x], s) & si\ x \in dom(ls) \\ (ls, s[v/x]) & autrement \end{cases}$$

La sémantique à petits pas des commandes est la suivante :

$$(f, \mathsf{skip}, S \cdot K, m) \to (f, S, K, m)$$

$$(f, x := e, K, m) \to (f, \mathsf{skip}, K, m[v/x]) \quad si\ (e, m) \Downarrow v$$

$$(f, S; S', K, m) \to (f, S, S' \cdot K, m)$$

$$(f, \mathsf{if}\ b\ \mathsf{then}\ S\ \mathsf{else}\ S', K, m) \to \begin{cases} (f, S, K, m) & si\ (b, m) \Downarrow \mathsf{true} \\ (f, S', K, m) & si\ (b, m) \Downarrow \mathsf{false} \end{cases}$$

$$(f, \mathsf{while}\ b\ \mathsf{do}\ S, K, s) \to \begin{cases} (f, S, (\mathsf{while}\ b\ \mathsf{do}\ S) \cdot K, m) & si\ (b, m) \Downarrow \mathsf{true} \\ (f, \mathsf{skip}, K, m) & si\ (b, m) \Downarrow \mathsf{false} \end{cases}$$

$$(f, x := \mathsf{call}\ g(e_1, \ldots, e_n), K, m) \to (g, S, \mathsf{returnto}(f, x, K, ls), m') \quad si\ (e_i, m) \Downarrow v_i, i = 1, \ldots, n, \\ g(x_1, \ldots, x_n) = \mathsf{var}\ x_{n+1}, \ldots, x_m; S, m = (ls, s), m' = (ls', s), \\ ls' = [v_1/x_1, \ldots, v_n/x_n, 0/x_{n+1}, \ldots, 0/x_m]$$

$$(f, \mathsf{return}(e), \mathsf{returnto}(g, x, K, ls'), m) \to (g, x := v, K, m') \quad si\ (e, m) \Downarrow v, m = (ls, s), m' = (ls', s)$$

$$(f, \mathsf{return}(e), S \cdot K, m) \to (f, \mathsf{return}(e), K, m)$$

Votre tâche *Dans la suite on donne la représentation en OCAML de la syntaxe abstraite et de l'évaluateur à petits pas du langage* Imp. *Proposez une extension qui couvre le langage* ImpF

```
(*** Syntaxe abstraite pour Imp ***)

type exp =
| Id of string
| Int of int
| Add of exp*exp ;;

type bexp = Less of exp*exp;;

type com =
| Skip
| Assign of string * exp
| Seq of com * com
| Ite of  bexp * com * com
| While of bexp * com ;;

type prog =
  Prog of com ;;

type cont =
  Halt
| Cons of com*cont ;;


(*** Mémoire ***)

type mem = (string*int) list;;

let rec value (m,x) = match m with
(y,v)::m' -> if x=y then v else value(m',x)
| _ -> failwith "undefined memory";;

(*** Mise à jour de la mémoire ***)

let rec update(m,x,v) = match m with
[] -> [(x,v)]
| (y,v1)::m' -> if x=y then (x,v)::m'
                       else (y,v1):: update(m',x,v);;
```

```
(*** Evaluaton expressions  ***)

let rec eval_exp (e,m) =
match e with
  Id(x) -> value(m,x)
| Int(x) -> x
| Add(e1,e2) ->  let x1= eval_exp (e1,m) in let x2= eval_exp (e2,m) in x1+x2 ;;

let eval_bool (b,m) = match b with
Less(e,e') -> let x=eval_exp(e,m) in let x'=eval_exp(e',m) in (x< x');;

(* Petits pas pour les commandes *)

let step (s,k,m) = match s with
Skip -> (match k with Cons(s',k') -> (s',k',m) | _ -> (s,k,m) )
| Assign(x,e) -> let v=eval_exp (e,m) in (Skip, k, update(m,x,v))
| Seq(s1,s2) -> (s1, Cons(s2,k),m)
| Ite(b,s1,s2) -> let x=eval_bool(b,m) in
                (if x then (s1,k,m) else (s2,k,m))
| While(b,s1) -> let x=eval_bool(b,m) in
                if x then (s1,Cons(While(b,s1),k),m)
                     else (Skip,k,m) ;;

(*** Execution ***)

let rec run (s,k,m) = match (s,k) with
(Skip, Halt) -> m
| _  -> let (s1,k1,m1) = step(s,k,m) in run(s1,k1,m1) ;;

let run_prog (p,m) = match p with
Prog(s) ->  run(s,Halt,m) ;;
```