École Normale Supérieure

Langages de programmation et compilation

examen 2013–2014

Jean-Christophe Filliâtre

16 janvier 2014

Les notes de cours manuscrites ou reprographiées sont les seuls documents autorisés.
Les deux problèmes sont indépendants.

---

1 Initialisation des variables locales

Dans le langage Java, une variable locale doit être initialisée avant d'être utilisée. Dans cet exercice, on se propose de vérifier cette condition. On considère la syntaxe abstraite suivante d'un fragment simplifié de Java, où e désigne une expression et s une instruction.

e	::=	x	variable
	|	$op(e, \ldots, e)$	opération (arithmétique, etc.)
s	::=	x `=` e	affectation
	|	`int` x	introduction d'une variable locale, non initialisée
	|	`if(` e `)` s `else` s	conditionnelle
	|	`while(` e `)` s	boucle
	|	s `;` s	séquence

On suppose que l'analyse de portée a déjà été effectuée et que deux variables distinctes portent des noms distincts.

Question 1 Quel peut être l'intérêt du compilateur de savoir que toute variable locale est initialisée avant d'être utilisée ?

Correction : Pas besoin d'initialiser les portions de mémoire correspondantes (dans le tableau d'activation, en l'occurrence).

Question 2 Soit S un ensemble de variables. On introduit le jugement noté $S \vdash e$ signifiant « l'expression e n'utilise que des variables apparaissant dans S ». Donner des règles d'inférence pour le jugement $S \vdash e$.

Correction : Il suffit de vérifier $fv(e) \subseteq S$ c'est-à-dire

$$\frac{x \in S}{S \vdash x} \qquad \frac{S \vdash e_1 \quad \ldots \quad S \vdash e_n}{S \vdash op(e_1, \ldots, e_n)}$$

Question 3 Pour vérifier qu'une instruction s est correcte, on introduit le jugement $S \vdash s \to S'$ signifiant « les variables de S étant supposées déjà initialisées, toute exécution de l'instruction s n'accède qu'à des variables déjà initialisées et, à l'issue de cette exécution, les variables de S' sont toutes initialisées ». En particulier, on a $S \subseteq S'$.

Donner des règles d'inférence pour le jugement $S \vdash s \to S'$.

Correction :

$$\frac{S \vdash e}{S \vdash x{=}e; \to S \cup \{x\}} \qquad \frac{}{S \vdash \texttt{int}\ x; \to S}$$

$$\frac{S \vdash e \quad S \vdash s_1 \to S_1 \quad S \vdash s_2 \to S_2}{S \vdash \texttt{if(}\ e\ \texttt{)}\ s_1\ \texttt{else}\ s_2 \to S_1 \cap S_2}$$

$$\frac{S \vdash e \quad S \vdash s \to S'}{S \vdash \texttt{while(}\ e\ \texttt{)}\ s \to S} \qquad \frac{S \vdash s_1 \to S_1 \quad S_1 \vdash s_2 \to S_2}{S \vdash s_1\ ;\ s_2 \to S_2}$$

Question 4 Réaliser l'analyse ci-dessus dans le langage OCaml, avec les types suivants pour la syntaxe abstraite :

```
type expr =
  | Var of string
  | App of string * expr list
type stmt =
  | Assign   of string * expr
  | Decl     of string
  | If       of expr * stmt * stmt
  | While    of expr * stmt
  | Sequence of stmt * stmt
```

Expliquer les différents éléments éventuellement introduits (types, fonctions, exceptions, etc.).

Correction :

```
module S = Set.Make(String)
type vars = S.t

exception NotInit of string

let rec expr s = function
  | Var x       -> if not (S.mem x s) then raise (NotInit x)
  | App (_, el) -> List.iter (expr s) el

let rec stmt s = function
  | Assign (x, e)       -> expr s e; S.add x s
  | Decl _              -> s
  | If (e, st1, st2)    -> expr s e; S.inter (stmt s st1) (stmt s st2)
  | While (e, st)       -> expr s e; ignore (stmt s st); s
  | Sequence (st1, st2) -> stmt (stmt s st1) st2
```

Question 5 Une telle analyse serait-elle aussi simple pour un langage tel que `C++` plutôt que le langage Java ?

Correction : Non. Le passage par référence permet d'initialiser une variable en la passant à une fonction (qui prend une référence en argument). De même avec un pointeur.

Note : cette question était initialement formulée comme « Une telle analyse serait-elle aussi simple dans le langage `C++` », ce qui est ambigu (s'agit-il du langage analysé ou du langage dans lequel est écrit l'analyse ?). La correction en a tenu compte, bien entendu.

---

2 Forme A-normale et style CPS

On considère ici un petit langage de programmation très simple, ne manipulant que des entiers. La syntaxe abstraite des expressions de ce langage est la suivante :

e	::=	n	constante entière
	$\mid$	x	variable
	$\mid$	`let` $x = e$ `in` e	déclaration locale
	$\mid$	`ifzero` e `then` e `else` e	conditionnelle
	$\mid$	$f(e, \ldots, e)$	appel de fonction

Comme on l'imagine, la construction `ifzero` teste si la première expression vaut 0. Un programme est un ensemble de fonctions mutuellement récursives et une expression servant de programme principal. La syntaxe abstraite est la suivante :

p	::=	`let rec`	$f(x, \ldots, x) = e$	définitions de fonctions
			$\vdots$	
		`and`	$f(x, \ldots, x) = e$	
		`in` e		programme principal

On suppose qu'il existe une fonction prédéfinie, `add`, qui calcule l'addition de deux entiers. On peut alors écrire le programme suivant pour calculer la factorielle de 10 :

```
let rec mul(x, y) = ifzero x then 0 else add(y, mul(add(x, -1), y))
    and fact(n)   = ifzero n then 1 else mul(n, fact(add(n, -1)))
in  fact 10
```

Question 6 Donner une sémantique opérationnelle à grand pas pour les expressions de ce langage, qui traduise un appel par valeur, sous la forme d'un jugement $e \xrightarrow{v} n$. On supposera donné un ensemble Δ de définitions de fonctions.

Correction :

$$\frac{}{n \xrightarrow{v} n} \qquad \frac{e_1 \xrightarrow{v} n_1 \quad e_2[x \leftarrow n_1] \xrightarrow{v} n_2}{\texttt{let } x = e_1 \texttt{ in } e_2 \xrightarrow{v} n_2}$$

$$\frac{e_1 \xrightarrow{v} 0 \quad e_2 \xrightarrow{v} n_2}{\texttt{ifzero } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \xrightarrow{v} n_2} \qquad \frac{e_1 \xrightarrow{v} n \neq 0 \quad e_3 \xrightarrow{v} n_3}{\texttt{ifzero } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \xrightarrow{v} n_3}$$

$$\frac{e_1 \xrightarrow{v} n_1 \quad e_2 \xrightarrow{v} n_2}{\texttt{add}(e_1, e_2) \xrightarrow{v} n_1 + n_2} \qquad \frac{f(x_1, \ldots, x_n) = e \in \Delta \quad e_i \xrightarrow{v} n_i \quad e[x_i \leftarrow n_i] \xrightarrow{v} n}{f(e_1, \ldots, e_n) \xrightarrow{v} n}$$

Forme A-normale. Dans la suite, un *atome* désigne une constante entière ou une variable. Une expression est dite en *forme A-normale* si tout appel de fonction est de la forme $f(a_1, \ldots, a_n)$, où les a_i sont des atomes, et si toute construction `ifzero` est de la forme `ifzero` a `then` e_1 `else` e_2 où a est un atome.

Question 7 Donner un programme en forme A-normale pour le calcul de la factorielle de 10.

Correction :

```
let rec mul(x, y) =
  ifzero x then 0
  else let p = add(x, -1) in
       let m = mul(p, y) in
       add(y, m)
and fact(n) =
  ifzero n then 1
  else let p = add(n, -1) in
       let r = fact(p) in
       mul(n, r)
in
fact 10
```

Question 8 Montrer que tout programme peut être transformé en un programme équivalent où toutes les expressions sont en forme A-normale.

Correction : On définit cette transformation A par

$$\begin{aligned} A(f(e_1, \ldots, e_n)) &= \texttt{let } x_1 = A(e_1) \texttt{ in } \ldots \\ &\quad\ \texttt{let } x_n = A(e_n) \texttt{ in} \\ &\quad\ f(x_1, \ldots, x_n) \\ A(\texttt{ifzero } e_1 \texttt{ then } e_2 \texttt{ else } e_3) &= \texttt{let } x_1 = A(e_1) \texttt{ in} \\ &\quad\ \texttt{ifzero } x_1 \texttt{ then } A(e_2) \texttt{ else } A(e_3) \end{aligned}$$

et un morphisme pour les autres constructions.

Par la suite, on ne considère plus que des programmes en forme A-normale.

Style CPS. Le style de programmation par continuation (en anglais CPS, pour *continuation passing style*) est une façon de programmer, dans un langage fonctionnel, où une fonction ne renvoie pas le résultat de son calcul mais le passe à une autre fonction, appelée *continuation*, qu'elle a reçue sous la forme d'un argument supplémentaire. Intuitivement, la continuation représente la suite des calculs que le programme doit effectuer.

Pour transformer nos programmes dans le style CPS, on considère un langage cible qui est une variante du langage mini-ML où les expressions (notées ici t) obéissent à la syntaxe abstraite suivante :

atome	a	::=	$n \mid x$	
expression	t	::=	a	atome
		|	`ifzero` a `then` t `else` t	conditionnelle
		|	$t(a, \ldots, a)$	application
		|	`fun` $x \rightarrow t$	fonction anonyme

En supposant une fonction prédéfinie `add` en style CPS dans ce langage cible, voici comment écrire en style CPS la fonction `mul` donnée plus haut :

```
let rec mul(x, y, k) =
  ifzero x then k(0)
  else add(x, -1, fun p -> mul(p, y, fun m -> add(y, m, k)))
```

La variable `k` est la continuation de la fonction `mul`. Dans la branche `then`, on souhaite renvoyer la valeur 0 et on la passe donc à la continuation, soit `k(0)`. Dans la branche `else`, on doit commencer par calculer `x-1`. Pour cela, on appelle la fonction `add` avec les deux arguments `x` et `-1` et une continuation qui effectuera la suite du calcul, à savoir multiplier le résultat par `y` puis encore après ajouter `y` et finalement passer le résultat à la continuation `k`.

D'une manière générale, on traduit une expression e du langage source, pour une continuation k donnée, en une expression $[e]_k$ du langage cible de la manière suivante :

$$\begin{aligned}
[n]_k &= k(n) \\
[x]_k &= k(x) \\
[\texttt{let } x = e_1 \texttt{ in } e_2]_k &= [e_1]_{(\texttt{fun } x \rightarrow [e_2]_k)} \\
[\texttt{ifzero } a_1 \texttt{ then } e_2 \texttt{ else } e_3]_k &= \texttt{ifzero } a_1 \texttt{ then } [e_2]_k \texttt{ else } [e_3]_k \\
[f(a_1, \ldots, a_n)]_k &= f(a_1, \ldots, a_n, k)
\end{aligned}$$

Par ailleurs, chaque définition de fonction $f(x_1, \ldots, x_n) = e$ du programme source est transformée en une définition de fonction du langage cible en lui ajoutant un argument qui est la continuation, c'est-à-dire

$$f(x_1, \ldots, x_n, k) = [e]_k$$

Enfin le programme principal est transformé avec la continuation identité, c'est-à-dire `fun` $x \rightarrow x$.

Question 9 Donner le résultat de la transformation en style CPS du programme suivant :

```
let rec fib(n) =
  ifzero n then 0 else
  let p = add(n, -1) in
  ifzero p then 1 else
  let y = fib(p) in
  let q = add(p, -1) in
  let z = fib(q) in
```

```
  add(y, z)
in
fib(10)
```

Correction :

```
let rec fib(n, k) =
  ifzero n then k(0) else
  add(n, -1, fun p ->
  ifzero p then k 1 else
  fib(p,     fun y ->
  add(p, -1, fun q ->
  fib(q,     fun z ->
  add(y, z, k)))))
in
fib(10, fun x -> x)
```

Question 10 Si une expression e est de type τ (en supposant qu'on a d'autres types que les entiers), quel sont les types de k et de $[e]_k$?

Correction : k est de type $\tau \to \tau'$ et $[e]_k$ de type τ', où τ' est le type de la valeur finale.

Question 11 Montrer que, dans un programme en style CPS, tout appel est un appel terminal. Quel est l'intérêt du point de vue de la compilation ?

Correction : Par récurrence structurelle sur e : dans les cas n, x et appel de fonction, le résultat est réduit à un appel terminal ; dans les cas `let` et `ifzero`, c'est immédiat par hypothèse de récurrence.
Intérêt : plus besoin de pile (et en particulier plus besoin de sauvegarder `$ra`).

Question 12 On considère le programme suivant en style CPS qui calcule la factorielle de 10 en supposant la fonction `mul` prédéfinie et en style CPS :

```
let rec fact(n, k) =
  ifzero n then k(1) else
  add (n, -1, fun p ->
  fact(p,     fun r ->
  mul (n, r,  k)))
in
fact(10, fun x -> x)
```

Donner le résultat de l'explicitation des fermetures (*closure conversion*, cf cours 8) sur ce programme, c'est-à-dire un programme où chaque fonction est maintenant de la forme

$$\texttt{letfun}\ f\ [y_1, \ldots, y_m](x_1, \ldots, x_n) = e$$

où les variables y_i sont les variables capturées dans la fermeture et les variables x_i sont les arguments, et où l'expression e ne contient plus de construction `fun` $x \rightarrow e$ mais une opération explicite de construction de fermeture `clos` $f\ [e_1, \ldots, e_m]$.

Correction :

```
letfun fun2 [n, k] (r) =
  mul(n, r, k)
letfun fun1 [n, k] (p) =
  fact(p, clos fun2 [n, k])
letfun fact [] (n, k) =
  ifzero n then k(1) else add(n, -1, clos fun1 [n, k])
letfun fun3 [] (x) =
  x
fact(10, clos fun3 [])
```

Note : on n'est pas obligé d'inclure `fact` dans la fermeture de `fun1` si on traite `fact` comme une fonction globale.

Question 13 Donner un code MIPS pour le programme obtenu à la question précédente, en optimisant tous les appels terminaux. On adoptera les conventions d'appel suivantes :

- pour une fonction comme `add`, `mul` ou `fact`, les arguments sont passés dans les registres `$a0`, `$a1`, `$a2` (il n'y en a jamais plus de trois) ;
- une fermeture est un pointeur vers un bloc alloué sur le tas contenant un pointeur vers le code à exécuter dans le premier champ et les valeurs des variables $y_1, \ldots, y_m$ dans les m champs suivants ;
- pour une application de fermeture, le code (de la fermeture) attend la fermeture dans le registre `$a0` et l'argument dans le registre `$a1` (il y a toujours un seul argument).

On ne demande pas d'écrire le code des fonctions `add` et `mul`. Un aide-mémoire MIPS est donné à la fin du sujet.

Correction :

Un peu long, certes, mais on ne demandait ni l'affichage dans `main`, ni le code de `add` et `mul`, ce qui fait seulement 42 lignes de MIPS au final.

```
        .text
main:   ## fact(10, clos fun3 [])
        li      $a0, 10
        la      $a1, clos3
        jal     fact
        ## impression du résultat final, pas demandé
        ## (aurait pu également être fait dans la continuation passée à fact)
        move    $a0, $v0
        li      $v0, 1          # print_int
        syscall
        li      $v0, 11         # print_char
        li      $a0, 10         # '\n'
        syscall
        li      $v0, 10         # exit
```

```
        syscall

##   letfun fact [] (n:$a0, k:$a1) =
##     ifzero n then k(1) else add(n, -1, clos fun1 [n, k])
fact:   bnez    $a0, L1
        ## then k(1)
        move    $a0, $a1
        li      $a1, 1
        lw      $t0, 0($a0)
        jr      $t0
L1:     ## else add(n, -1, clos fun1 [n, k])
        move    $s0, $a0        # sauve n dans s0
        move    $s1, $a1        # sauve k dans s1
        li      $a0, 12         # alloue une fermeture, dans $a2
        li      $v0, 9
        syscall
        move    $a2, $v0
        la      $t0, fun1       # la remplie avec fun1,n,k
        sw      $t0, 0($a2)
        sw      $s0, 4($a2)
        sw      $s1, 8($a2)
        move    $a0, $s0        # appel add(n, -1, clos ...)
        li      $a1, -1
        j       add1            # terminal (pourrait même être omis !)

## add(x:$a0, y:$a1, k:$a2)
add1:   add     $a1, $a0, $a1
        move    $a0, $a2
        lw      $t0, 0($a2)
        jr      $t0             # terminal

## mul(x:$a0, y:$a1, k:$a2)
mul1:   mul     $a1, $a0, $a1
        move    $a0, $a2
        lw      $t0, 0($a2)
        jr      $t0             # terminal

##   letfun fun1 [n, k] (p:$a1) =
##     fact(p, clos fun2 [n, k])
fun1:   move    $s0, $a0        # sauve la fermeture dans s0
        move    $s1, $a1        # sauve p dans s1
        li      $a0, 12         # alloue une nouvelle fermeture, dans $a1
        li      $v0, 9          # (note : on pourrait réutiliser la même fermeture)
        syscall
        move    $a1, $v0
        la      $t0, fun2       # la remplie avec fun2,n,k
        sw      $t0, 0($a1)
        lw      $t0, 4($s0)
        sw      $t0, 4($a1)
        lw      $t0, 8($s0)
        sw      $t0, 8($a1)
```

```
        move    $a0, $s1        # appel fact(p, clos ...)
        j       fact            # terminal

##   letfun fun2 [n, k] (r) =
##     mul(n, r, k)
fun2:   move    $s0, $a0
        lw      $a0, 4($s0)
        lw      $a2, 8($s0)
        j       mul1            # terminal

##   letfun fun3 [] (x) =
##     x
fun3:   move    $v0, $a1
        jr      $ra             # c'est le seul endroit où on utilise $ra

        .data
# la fermeture de fun3 peut être allouée statiquement (toujours l'identité)
clos3:  .word   fun3
```

Question 14 Qu'est devenue la pile qui est traditionnellement utilisée pour exécuter la version récursive de la fonction factorielle ?

Correction : Elle est allouée sur le tas, sous la forme de fermetures. Ces fermetures forment une liste chaînée, contenant les valeurs n, $n-1$, $n-2$, etc. (chaînées dans le sens croissant). Et une liste chaînée n'est rien d'autre qu'une pile.

Annexe : aide-mémoire MIPS

On donne ici un fragment du jeu d'instructions MIPS. Vous êtes libre d'utiliser tout autre élément de l'assembleur MIPS. Dans ce qui suit, r_i désigne un registre, n une constante entière et L une étiquette.

`li` r_1, n	charge la constante n dans le registre r_1
`la` r_1, L	charge l'adresse de l'étiquette L dans le registre r_1
`addi` r_1, r_2, n	calcule la somme de r_2 et n dans r_1
`add` r_1, r_2, r_3	calcule la somme de r_2 et r_3 dans r_1 (on a de même `sub`, `mul` et `div`)
`move` r_1, r_2	copie le registre r_2 dans le registre r_1
`lw` r_1, $n(r_2)$	charge dans r_1 la valeur contenue en mémoire à l'adresse $r_2 + n$
`sw` r_1, $n(r_2)$	écrit en mémoire à l'adresse $r_2 + n$ la valeur contenue dans r_1
`beq` r_1, r_2, L	saute à l'adresse désignée par l'étiquette L si $r_1 = r_2$ (on a de même `bne`, `blt`, `ble`, `bgt` et `bge`)
`beqz` r_1, L	saute à l'adresse désignée par l'étiquette L si $r_1 = 0$ (on a de même `bnez`, `bltz`, `blez`, `bgtz` et `bgez`)
`j` L	saute à l'adresse désignée par l'étiquette L
`jr` r_1	saute à l'adresse contenue dans le registre r_1
`jal` L	saute à l'adresse désignée par l'étiquette L, après avoir sauvegardé l'adresse de retour dans `$ra`
`jalr` r_1	saute à l'adresse contenue dans le registre r_1, après avoir sauvegardé l'adresse de retour dans `$ra`