École Polytechnique

INF564 : Compilation

examen 2018 (X2015)

Jean-Christophe Filliâtre

19 mars 2018

Les notes de cours manuscrites ou reprographiées sont les seuls documents autorisés.
L'épreuve dure 3 heures.

Dans tout ce sujet, on considère un petit langage de programmation avec des entiers et des paires. Un programme est composé d'un ensemble de fonctions mutuellement récursives et d'une expression principale. Chaque fonction a un unique argument, toujours appelé `x`. Voici un programme dans ce langage, calculant la factorielle de 10 :

```
def mult(x) =
  ifzero fst(x) then 0 else add((snd(x), mult((add((fst(x),-1)), snd(x)))))
def fact(x) =
  ifzero x then 1 else mult((x, fact(add((x, -1)))))
fact(10)
```

Ce langage est muni d'une sémantique d'appel par valeur. Il y a trois fonctions prédéfinies :
- `add` : attend une paire d'entiers en argument et renvoie leur somme ;
- `fst` : attend une paire en argument et renvoie sa première composante ;
- `snd` : attend une paire en argument et renvoie sa seconde composante.

La syntaxe abstraite de ce langage est donnée figure 1.

Les différentes parties du sujet sont indépendantes. Néanmoins, elles nécessitent d'avoir bien compris la sémantique de ce langage décrite dans la partie 1.

1 Sémantique

On munit ce langage d'une sémantique opérationnelle à petits pas de la forme

$$e \to e'$$

où e et e' sont deux expressions. Les règles de la sémantique opérationnelle sont données figure 2. La notation $e[\mathtt{x} \leftarrow v]$ désigne l'expression obtenue en remplaçant dans e toute occurrence de `x` par v. On prendra le temps de bien comprendre les règles de la sémantique. L'*évaluation* d'une expression e est une séquence de réductions partant de e et conduisant à une valeur, c'est-à-dire

$$e \to e_1 \to e_2 \cdots \to v.$$

L'évaluation d'un programme est l'évaluation de son expression principale.

Question 1 Donner l'évaluation du programme

```
def f(x) = add((x, x))
f(add((add((3,5)), add((5,8)))))
```

e	::=	**expression**
	$\mid\ n$	constante entière $n \in \mathbb{Z}$
	$\mid$ `x`	variable
	$\mid$ $(e\,,\,e)$	construction d'une paire
	$\mid$ $f(e)$	appel de fonction
	$\mid$ `ifzero` e `then` e `else` e	conditionnelle
d	::=	**définition de fonction**
	$\mid$ `def` f`(x) =` e	
p	::=	**programme**
	$\mid$ $d \ldots d\ e$	

FIGURE 1 – Syntaxe abstraite.

v	::=	**valeur**
	$\mid\ n$	constante entière
	$\mid\ (v\,,\,v)$	paire

$$\frac{e_1 \to e_1'}{(e_1\,,\,e_2) \to (e_1'\,,\,e_2)} \qquad \frac{e_2 \to e_2'}{(v_1\,,\,e_2) \to (v_1\,,\,e_2')}$$

$$\frac{e_1 \to e_1'}{f(e_1) \to f(e_1')} \qquad \frac{\texttt{def}\ f\texttt{(x) =}\ e}{f(v) \to e[\texttt{x} \leftarrow v]}$$

$$\frac{n = n_1 + n_2}{\texttt{add}((n_1\,,\,n_2)) \to n} \qquad \frac{}{\texttt{fst}((v_1\,,\,v_2)) \to v_1} \qquad \frac{}{\texttt{snd}((v_1\,,\,v_2)) \to v_2}$$

$$\frac{e_1 \to e_1'}{\texttt{ifzero}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \to \texttt{ifzero}\ e_1'\ \texttt{then}\ e_2\ \texttt{else}\ e_3}$$

$$\frac{}{\texttt{ifzero}\ 0\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \to e_2} \qquad \frac{v \neq 0}{\texttt{ifzero}\ v\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \to e_3}$$

FIGURE 2 – Sémantique opérationnelle à petits pas.

Question 2 Donner un programme qui n'a pas d'évaluation.

Listes. On peut facilement représenter une liste contenant n valeurs $x_1, \ldots, x_n$ avec des paires, sous la forme

$$(x_1, (x_2, \ldots, (x_n, 0) \ldots))$$

c'est-à-dire exactement une paire pour chaque élément de la liste et l'entier 0 pour représenter la fin de la liste. On peut alors définir des fonctions récursives sur de telles listes. Voici par exemple une fonction `len` qui renvoie la longueur d'une liste et une fonction `rev` qui renverse une liste :

```
def len(x) =
  ifzero x then 0 else add((1, len(snd(x))))
def rev_aux(x) =
  ifzero fst(x) then snd(x) else rev_aux((snd(fst(x)), (fst(fst(x)), snd(x))))
def rev(x) =
  rev_aux((x, 0))
```

Question 3 Définir une fonction `fib` telle que, pour tout entier $n \geq 1$, l'évaluation de `fib(`n`)` renvoie la liste des $n+1$ premiers entiers de la suite de Fibonacci, dans l'ordre croissant. On rappelle que la suite de Fibonacci est définie par

$$\begin{cases} F_0 & = & 0 \\ F_1 & = & 1 \\ F_{n+2} & = & F_n + F_{n+1} \quad \text{pour } n \geq 0 \end{cases}$$

La complexité, définie comme le nombre total de petits pas, doit être $O(n)$.

2 Interprète

On souhaite réaliser un interprète de notre langage, c'est-à-dire un programme qui reçoit en argument la syntaxe abstraite d'un programme de notre langage et qui

- affiche la valeur à laquelle aboutit l'évaluation de ce programme, le cas échéant ;
- signale une erreur si la séquence de réductions aboutit à une expression qui n'est pas une valeur et qui ne se réduit pas ;
- ne termine pas sinon.

Question 4 Donner les différents types, structures de données et fonctions/méthodes composant cet interprète, en OCaml ou en Java (au choix). On ne demande pas en revanche d'écrire le *code* de l'interprète.

3 Compilation vers x86-64

On se propose maintenant de compiler notre langage vers l'assembleur x86-64. On adopte le schéma de compilation suivant :

- une valeur est soit un entier signé 64 bits, soit un pointeur vers un bloc de 16 octets sur le tas contenant les deux composantes d'une paire (juxtaposées) ;
- la valeur d'une expression est calculée dans `%rax` ;
- le valeur de `x`, le cas échéant, est contenue dans `%rdi`.

La figure 3 contient une partie du compilateur. Une expression e est compilée par un appel à *compile*(e). Un aide-mémoire x86-64 est donné en annexe.

$$
\begin{array}{rcl}
\mathit{compile}(n) & = & \texttt{movq \$}n\texttt{, \%rax} \\
\mathit{compile}(\texttt{x}) & = & \texttt{movq \%rdi, \%rax} \\
\mathit{compile}(\texttt{(}e_1\texttt{, }e_2\texttt{)}) & = & \ldots \\
\mathit{compile}(f\texttt{(}e_1\texttt{)}) & = & \ldots \\
\mathit{compile}(\texttt{ifzero } e_1 \texttt{ then } e_2 \texttt{ else } e_3) & = & \ldots \\
 & & \\
\mathit{compile}(\texttt{def } f\texttt{(x) = } e) & = & f : \\
 & & \mathit{compile}(e) \\
 & & \texttt{ret}
\end{array}
$$

FIGURE 3 – Compilation vers x86-64.

Question 5 Donner le code du compilateur pour
- la construction de la paire $\texttt{(}e_1\texttt{, }e_2\texttt{)}$;
- la conditionnelle $\texttt{ifzero } e_1 \texttt{ then } e_2 \texttt{ else } e_3$;
- l'application $f\texttt{(}e_1\texttt{)}$.

Question 6 Donner le code assembleur des fonctions prédéfinies `fst`, `snd` et `add`.

Question 7 Donner le code assembleur d'une fonction `print_list` qui reçoit en argument (dans `%rdi`) une valeur supposée être une liste d'entiers et affiche ses éléments dans l'ordre. On suppose l'existence d'une fonction `print_int` pour afficher un entier, avec les conventions d'appel usuelles.

Question 8 On aimerait pouvoir écrire une fonction assembleur `print` qui affiche n'importe quelle valeur supposée bien formée, par exemple sous la forme

```
((1, 2), ((3, 4), 0))
```

Ce n'est malheureusement pas possible, car il faudrait être capable de faire la différence entre un entier et un pointeur (vers une paire). Proposer une solution pour y remédier.

4 Analyse syntaxique

On cherche maintenant à construire un analyseur syntaxique pour notre langage. On se donne la grammaire

$$
\begin{array}{rcl}
S & \to & E \\
E & \to & \texttt{int} \\
 & \mid & \texttt{x} \\
 & \mid & \texttt{(}\ E\ \texttt{,}\ E\ \texttt{)} \\
 & \mid & \texttt{ifzero } E \texttt{ then } E \texttt{ else } E \\
 & \mid & \texttt{ident (}\ E\ \texttt{)}
\end{array}
$$

où S est l'axiome et où les terminaux sont `int` (une constante entière), `x`, `(`, `,`, `)`, `ifzero`, `then`, `else` et `ident` (un identificateur autre que `x`, `ifzero`, `then` et `else`).

$$\frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{\Gamma = \texttt{x} : \tau}{\Gamma \vdash \texttt{x} : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

$$\frac{f : \tau_1 \to \tau_2 \quad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash f(e_1) : \tau_2} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \times \texttt{int}}{\Gamma \vdash \texttt{add}(e_1) : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{fst}(e_1) : \tau_1} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{snd}(e_1) : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{ifzero } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \tau}$$

FIGURE 4 – Typage.

Question 9 Montrer que cette grammaire est LR(0). Indication : on pourra construire l'automate déterministe directement ; il contient une vingtaine d'états.

Question 10 Que peut-on en déduire quant à l'analyse de cette grammaire par un outil tel que `cup` ou `menhir` ?

5 Typage

Dans cette partie, on va typer notre langage avec des types de la forme

τ	::=		**type**
		$\texttt{int}$	type d'un entier
		$\tau \times \tau$	type d'une paire

Un environnement de typage Γ est soit vide, soit réduit à $\texttt{x} : \tau$. On note $\Gamma \vdash e : \tau$ le jugement « dans l'environnement Γ, l'expression e a le type τ ». On note $f : \tau_1 \to \tau_2$ si la fonction f attend un argument de type τ_1 et renvoie un résultat de type τ_2. Les règles de typage des expressions sont données figure 4. Un programme est typable s'il existe un type pour chacune de ses fonctions de telle sorte que

— si le type donné à f est $\tau_1 \to \tau_2$ et `def` f`(x) =` e alors $\texttt{x} : \tau_1 \vdash e : \tau_2$;
— l'expression principale est bien typée dans l'environnement vide.

Question 11 Pour chacun des trois programmes suivants, indiquer s'il est typable. Si oui, donner un type à sa fonction. Sinon, justifier qu'il n'est pas typable.

1. ```
   def oups(x) = oups(x)
   oups(0)
   ```
2. ```
   def len(x) = ifzero x then 0 else add((1, len(snd(x))))
   len((1, (0, 0)))
   ```
3. ```
   def foo(x) = ifzero fst(x) then x else foo((fst(snd(x)), (snd(snd(x)), fst(x))))
   foo((1,(2,0)))
   ```

Sûreté du typage. Comme vu en cours, la sûreté du typage se déduit des résultats de progrès et de préservation, qui font l'objet des questions suivantes.

Question 12 Montrer la propriété de progrès : si $\vdash e : \tau$, alors soit e est une valeur, soit il existe e' telle que $e \to e'$.
   ```

Question 13 Montrer que si $\mathtt{x} : \tau_1 \vdash e_2 : \tau_2$ et $\vdash e_1 : \tau_1$ alors $\vdash e_2[\mathtt{x} \leftarrow e_1] : \tau_2$.

Question 14 Montrer la propriété de préservation du typage : si $\vdash e : \tau$ et $e \to e'$, alors $\vdash e' : \tau$.

Question 15 En déduire que, si $\vdash e : \tau$, alors la réduction de e est infinie ou termine sur une valeur.

Annexe : aide-mémoire x86-64

On donne ici un fragment du jeu d'instructions x86-64. Vous êtes libre d'utiliser tout autre élément de l'assembleur x86-64. Dans ce qui suit, r_i désigne un registre, n une constante entière et L une étiquette.

`mov` r_2`,` r_1	copie le registre r_2 dans le registre r_1
`mov` `$`n`,` r_1	charge la constante n dans le registre r_1
`add` r_2`,` r_1	calcule $r_1 + r_2$ et l'affecte à r_1
`mov` n`(`r_2`),` r_1	charge dans r_1 la valeur contenue en mémoire à l'adresse $r_2 + n$
`mov` r_1`,` n`(`r_2`)`	écrit en mémoire à l'adresse $r_2 + n$ la valeur contenue dans r_1
`push` r_1	empile la valeur contenue dans r_1
`pop` r_1	dépile une valeur dans le registre r_1
`test` r_2`,` r_1	positionne les drapeaux en fonction de la valeur de r_1 ET r_2
`jz` L	saute à l'adresse désignée par l'étiquette L si les drapeaux signalent un résultat nul
`jmp` L	saute à l'adresse désignée par l'étiquette L
`call` L	saute à l'adresse désignée par l'étiquette L, après avoir empilé l'adresse de retour
`ret`	dépile une adresse et y effectue un saut

On alloue de la mémoire sur le tas avec un appel à `malloc`, qui attend un nombre d'octets dans `%rdi` et renvoie l'adresse du bloc alloué dans `%rax`.