École Normale Supérieure

Langages de programmation et compilation

examen 2017–2018

Jean-Christophe Filliâtre

26 janvier 2018

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

Dans tout ce sujet, on considère une variante de mini-ML dont la syntaxe abstraite est la suivante :

e	::=	n	constante entière $n \in \mathbb{Z}$
		x	variable
	|	$e - e$	soustraction
	|	`fun` $x \to e$	fonction anonyme
	|	$e\ e$	application
	|	`print` e	affichage
	|	`if` e `<= 0 then` e `else` e	conditionnelle
	|	`fix` e	point fixe

Dans la suite, on pourra écrire `let` $x = e_1$ `in` e_2 comme un sucre syntaxique pour (`fun` $x \to e_2$) e_1. Dans ce langage, un programme est réduit à une expression.

Les parties 1 et 2 ne sont pas indépendantes, la partie 2 utilisant des définitions et des résultats de la partie 1. Les parties 3 et 4 sont indépendantes entre elles et des parties 1 et 2.

1 Sémantique

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

$$e \xrightarrow{a} e'$$

où e et e' sont deux expressions et a l'affichage réalisé par le pas de réduction. Cet affichage vaut soit $\emptyset$, lorsque rien n'est affiché, soit un entier n, lorsque le pas d'exécution affiche l'entier n. La sémantique opérationnelle à petits pas est donnée figure 1. On prendra le temps de bien la lire et notamment de relever les petites différences avec celle vue en cours. Une *évaluation* d'une expression e est une séquence de réductions qui conduit à une valeur, c'est-à-dire

$$e \xrightarrow{a_1} e_1 \xrightarrow{a_2} e_2 \cdots \xrightarrow{a_n} v.$$

Question 1 Donner *une* évaluation de l'expression

$$(\texttt{print}\ 2) - ((\texttt{print}\ 60) - (\texttt{print}\ 100))$$

Combien y a-t-il d'évaluations possibles pour cette expression ? Justifier.

valeurs

$$\begin{array}{rcll} v & ::= & n & \text{constante} \\ & | & \texttt{fun}\ x \to e & \text{fonction} \end{array}$$

affichage

$$\begin{array}{rcll} a & ::= & \emptyset & \text{on n'affiche rien} \\ & | & n & \text{on affiche } n \end{array}$$

réductions de tête

$$\begin{array}{rcll} n_1 - n_2 & \xrightarrow{\emptyset} & n & \text{avec } n \text{ l'entier } n_1 - n_2 \\ (\texttt{fun}\ x \to e)\ v & \xrightarrow{\emptyset} & e[x \leftarrow v] & \\ \texttt{print}\ n & \xrightarrow{n} & n & \\ \texttt{if}\ n\ \texttt{<=}\ \texttt{0}\ \texttt{then}\ e_1\ \texttt{else}\ e_2 & \xrightarrow{\emptyset} & e_1 & \text{si } n \leq 0 \\ \texttt{if}\ n\ \texttt{<=}\ \texttt{0}\ \texttt{then}\ e_1\ \texttt{else}\ e_2 & \xrightarrow{\emptyset} & e_2 & \text{si } n > 0 \\ \texttt{fix}\ (\texttt{fun}\ x \to e) & \xrightarrow{\emptyset} & e[x \leftarrow \texttt{fix}\ (\texttt{fun}\ x \to e)] & \end{array}$$

contextes de réduction

$$\begin{array}{rcl} E & ::= & \square \\ & | & E - e \\ & | & e - E \\ & | & E\ e \\ & | & e\ E \\ & | & \texttt{print}\ E \\ & | & \texttt{if}\ E\ \texttt{<=}\ \texttt{0}\ \texttt{then}\ e\ \texttt{else}\ e \\ & | & \texttt{fix}\ E \end{array}$$

réduction

$$\frac{e_1 \xrightarrow{a} e_2}{E(e_1) \xrightarrow{a} E(e_2)} \quad \text{pour } E \neq \square$$

FIGURE 1 – Sémantique opérationnelle à petits pas.

Question 2 Donner une expression e telle que, pour tout entier n, toute évaluation de $e\ n$ affiche exactement un entier, égal à la factorielle de n.

Confluence du calcul. On va montrer maintenant que le résultat d'un calcul ne dépend pas de l'ordre d'évaluation, tant qu'on ignore les affichages. Dans les deux questions suivantes, on s'autorise à écrire $e \to e'$ pour une réduction dont on ne précise pas l'affichage.

Question 3 Montrer que si $e \to e_l$ et $e \to e_r$ alors soit $e_l = e_r$, soit il existe e' telle que $e_l \to e'$ et $e_r \to e'$.

Question 4 En déduire que si $e \to e_1 \to e_2 \cdots \to e_n$ et $e \to e'_1 \to e'_2 \cdots \to e'_{n'}$ alors il existe e_m telle que $e_n \to e_{n+1} \to e_{n+2} \cdots \to e_m$ ($m-n$ étapes) et $e'_{n'} \to e'_{n'+1} \to e'_{n'+2} \cdots \to e_m$ ($m-n'$ étapes) avec $m \leq n + n'$.

2 Typage avec effets

Dans cette partie, on va typer notre langage avec des types enrichis d'une notion d'*effet*. Un effet, noté ϕ, prend trois valeurs possibles :

$$\begin{array}{rcll} \phi & ::= & \bot & \text{le calcul ne produit aucun affichage} \\ & | & \mathtt{P} & \text{le calcul peut produire un affichage, qui ne dépend pas de l'ordre d'évaluation} \\ & | & \top & \text{le calcul peut produire un affichage, qui peut dépendre de l'ordre d'évaluation} \end{array}$$

Les effets sont naturellement ordonnés de la manière suivante

$$\bot \leq \mathtt{P} \leq \top$$

et on s'autorisera à écrire $\max(\ldots)$ pour calculer le maximum de plusieurs effets. On définit par ailleurs une opération binaire $\sqcup$ sur les effets, de la manière suivante :

$$\begin{array}{rcll} \bot \sqcup \phi & \stackrel{\text{def}}{=} & \phi & \\ \phi \sqcup \bot & \stackrel{\text{def}}{=} & \phi & \\ \phi_1 \sqcup \phi_2 & \stackrel{\text{def}}{=} & \top & \text{sinon} \end{array}$$

Un environnement de typage, noté Γ, associe des types à des variables. Le jugement de typage prend la forme

$$\Gamma \vdash e : \tau \ \& \ \phi$$

et se lit comme « dans l'environnement Γ, l'expression e a le type τ et l'effet ϕ ». Les types sont ici des types simples, de la forme

$$\begin{array}{rcll} \tau & ::= & \mathtt{int} & \text{type des entiers} \\ & | & \tau \stackrel{\phi}{\to} \tau & \text{type des fonctions} \end{array}$$

Le type d'une fonction contient un effet au-dessus de la flèche, appelé *effet latent*. C'est l'effet de l'évaluation d'une application de cette fonction. Les règles de typage sont données figure 2.

$$\frac{}{\Gamma \vdash n : \texttt{int} \;\&\; \bot} \qquad \frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x) \;\&\; \bot} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \;\&\; \phi_1 \quad \Gamma \vdash e_2 : \texttt{int} \;\&\; \phi_2}{\Gamma \vdash e_1 - e_2 : \texttt{int} \;\&\; \phi_1 \sqcup \phi_2}$$

$$\frac{\Gamma + x : \tau_1 \vdash e : \tau_2 \;\&\; \phi}{\Gamma \vdash \texttt{fun}\ x \to e : \tau_1 \xrightarrow{\phi} \tau_2 \;\&\; \bot} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \xrightarrow{\phi} \tau_1 \;\&\; \phi_1 \quad \Gamma \vdash e_2 : \tau_2 \;\&\; \phi_2}{\Gamma \vdash e_1\ e_2 : \tau_1 \;\&\; \max(\phi_1 \sqcup \phi_2, \phi)}$$

$$\frac{\Gamma \vdash e : \texttt{int} \;\&\; \phi}{\Gamma \vdash \texttt{print}\ e : \texttt{int} \;\&\; \max(\phi, \texttt{P})} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \;\&\; \phi_1 \quad \Gamma \vdash e_2 : \tau \;\&\; \phi_2 \quad \Gamma \vdash e_3 : \tau \;\&\; \phi_3}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{<= 0 then}\ e_2\ \texttt{else}\ e_3 : \tau \;\&\; \max(\phi_1, \phi_2, \phi_3)}$$

$$\frac{\Gamma \vdash e : (\tau_1 \xrightarrow{\phi} \tau_2) \xrightarrow{\phi_2} (\tau_1 \xrightarrow{\phi_1} \tau_2) \;\&\; \phi_3 \quad \phi \stackrel{\text{def}}{=} \max(\phi_1, \phi_2)}{\Gamma \vdash \texttt{fix}\ e : \tau_1 \xrightarrow{\phi} \tau_2 \;\&\; \phi_3}$$

FIGURE 2 – Typage avec effets.

Question 5 Donner trois expressions e_1, e_2 et e_3 ayant respectivement les types suivants :

$$\vdash e_1 : \texttt{int} \;\&\; \top$$
$$\vdash e_2 : \texttt{int} \xrightarrow{\texttt{P}} \texttt{int} \;\&\; \texttt{P}$$
$$\vdash e_3 : (\texttt{int} \xrightarrow{\top} \texttt{int}) \xrightarrow{\texttt{P}} \texttt{int} \;\&\; \bot$$

Question 6 L'expression suivante est-elle typable ?

$$\texttt{fun}\ f \to f\ (\texttt{fun}\ x \to 0) - f\ (\texttt{fun}\ x \to \texttt{print}\ 0)$$

Si oui, lui donner un type. Sinon, justifier qu'elle n'est pas typable.

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 deux questions suivantes.

Question 7 Montrer la propriété de progrès : si $\vdash e : \tau \;\&\; \phi$, alors soit e est une valeur, soit il existe e' et a tels que $e \xrightarrow{a} e'$.

Question 8 Montrer la propriété de préservation du typage : si $\Gamma \vdash e : \tau \;\&\; \phi$ et $e \xrightarrow{a} e'$, alors $\Gamma \vdash e' : \tau \;\&\; \phi'$ avec $\phi' \leq \phi$.

On admettra la propriété de substitution : si $\Gamma + x : \tau' \vdash e : \tau \;\&\; \phi$ et $\Gamma \vdash e' : \tau' \;\&\; \bot$ alors $\Gamma \vdash e[x \leftarrow e'] : \tau \;\&\; \phi$.

Correction de l'effet $\bot$. On va maintenant chercher à montrer que si l'effet d'une expression est $\bot$, alors son évaluation ne produit aucun affichage.

Question 9 Montrer que si $\Gamma \vdash e : \tau \;\&\; \phi$ et $e \xrightarrow{n} e'$, alors $\phi \neq \bot$.

Question 10 Déduire du résultat précédent que, si $\Gamma \vdash e : \tau \;\&\; \phi$ et $e \xrightarrow{a_1} e_1 \xrightarrow{a_2} e_2 \cdots \xrightarrow{a_n} e_n$ et s'il existe i tel que $a_i \neq \emptyset$, alors $\phi \neq \bot$.

Correction de l'effet `P`. Enfin, on va chercher à montrer que si l'effet d'une expression est `P`, alors son évaluation produit un affichage qui ne dépend pas de l'ordre d'évaluation. Pour cela, on se donne une opération binaire $\cdot$ sur les affichages, associative et d'élément neutre $\emptyset$.

Question 11 Montrer que si $\Gamma \vdash e : \tau \ \& \ \phi$, avec $\phi \leq \texttt{P}$, et si $e \xrightarrow{a_l} e_l$ et $e \xrightarrow{a_r} e_r$, avec $e_l \neq e_r$, alors il existe e' telle que $e_l \xrightarrow{a'_l} e'$, $e_r \xrightarrow{a'_r} e'$ et $a_l \cdot a'_l = a_r \cdot a'_r$.

Question 12 Montrer que si $\Gamma \vdash e : \tau \ \& \ \phi$, avec $\phi \leq \texttt{P}$, et si $e \xrightarrow{a_1} e_1 \xrightarrow{a_2} e_2 \cdots \xrightarrow{a_n} e_n$ et $e \xrightarrow{a'_1} e'_1 \xrightarrow{a'_2} e'_2 \cdots \xrightarrow{a'_{n'}} e'_{n'}$, avec $e_n = e'_{n'} = v$ une valeur, alors $a_1 \cdot a_2 \cdots a_{n-1} \cdot a_n = a'_1 \cdot a'_2 \cdots a'_{n'-1} \cdot a'_{n'}$ et $n = n'$.

On pourra admettre que si $e \xrightarrow{a} e'$ et $e \xrightarrow{a'} e'$ alors $a = a'$.

3 Production de code

On se propose maintenant de compiler notre langage vers l'assembleur x86-64. Comme expliqué en cours, on procède en deux temps, en commençant par une étape d'*explicitation des fermetures*. Plus précisément, un programme e est traduit vers un ensemble de définitions de fonctions globales $d_1, \ldots, d_n$ d'une part et une expression c d'autre part, dans la syntaxe abstraite suivante :

$$
\begin{array}{rcl}
c & ::= & n \\
 & | & x \\
 & | & c - c \\
 & | & \texttt{clos}\ f\ [E] \\
 & | & c\ c \\
 & | & \texttt{print}\ c \\
 & | & \texttt{if}\ c\ \texttt{<= 0 then}\ c\ \texttt{else}\ c \\
 & | & \texttt{fix}\ c \\
d & ::= & \texttt{letfun}\ f\ [E]\ x\ \texttt{=}\ c \\
E & ::= & x, \ldots, x
\end{array}
$$

Dans ce langage, la construction `fun` $x \to e$ a été remplacée par une opération explicite de construction de fermeture `clos` $f\ [E]$ où f est maintenant une fonction globale et $E = x_1, \ldots, x_m$ est l'*environnement*, c'est-à-dire la liste des variables libres de `fun` $x \to e$. Cette liste est ordonnée de façon arbitraire. On note E_i le i-ième élément de la liste E, les éléments étant numérotés à partir de 1.

Question 13 Donner le résultat de l'explicitation des fermetures sur le programme

`print` $((\texttt{fix}\ (\texttt{fun}\ fib \to \texttt{fun}\ n \to \texttt{if}\ n-1\ \texttt{<= 0 then}\ n\ \texttt{else}\ (fib\ (n-1)) - (0 - (fib\ (n-2))))))\ 10)$

Assembleur. L'étape suivante consiste à traduire ce langage intermédiaire 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 une fermeture sur le tas (toutes les valeurs ont donc la même taille) ;
- une fermeture est un bloc alloué sur le tas contenant $n+1$ mots de 64 bits, le premier contenant une adresse de code et les suivants les valeurs de l'environnement ;
- la valeur d'une expression est calculée dans `%rax` ;
- dans le code d'une fonction :

$$
\begin{array}{rcll}
\mathit{compile}(E, n) & = & \texttt{movq \$}n\texttt{,\%rax} & \\
\mathit{compile}(E, x) & = & \texttt{movq \%rdi, \%rax} & \text{si } x \notin E \\
\mathit{compile}(E, x) & = & \texttt{movq } 8i\texttt{(\%rsi), \%rax} & \text{si } x = E_i \\
\mathit{compile}(E, c_1 - c_2) & = & \ldots & \\
\mathit{compile}(E, \texttt{clos } f\ [x_1, \ldots, x_n]) & = & \ldots & \\
\mathit{compile}(E, c_1\ c_2) & = & \ldots & \\
\mathit{compile}(E, \texttt{print } c) & = & \ldots & \\
\mathit{compile}(E, \texttt{if } c_1 \texttt{ <= 0 then } c_2 \texttt{ else } c_3) & = & \ldots & \\
\mathit{compile}(E, \texttt{fix } c) & = & \mathit{compile}(E, c) & \\
 & & \texttt{pushq \%rax} & \\
 & & \texttt{pushq \%rdi} & \\
 & & \texttt{pushq \%rsi} & \\
 & & \texttt{movq \$16, \%rdi} & \\
 & & \texttt{call malloc} & \\
 & & \texttt{popq \%rsi} & \\
 & & \texttt{popq \%rdi} & \\
 & & \texttt{movq \$\_fix, (\%rax)} & \\
 & & \texttt{popq \%rcx} & \\
 & & \texttt{movq \%rcx, 8(\%rax)} & \\
 & & & \\
\mathit{compile}(\texttt{letfun } f\ E\ x \texttt{ = } c) & = & f : & \\
 & & \mathit{compile}(E, c) & \\
 & & \texttt{ret} &
\end{array}
$$

FIGURE 3 – Compilation vers x86-64.

  - la valeur de l'argument est contenue dans `%rdi`,
  - la valeur de la fermeture est contenue dans `%rsi` ;
- `%rcx` est utilisé comme temporaire.

La figure 3 contient une partie du compilateur. Une expression c est compilée par un appel à $\mathit{compile}(E, c)$ où E est l'environnement de la fermeture, lorsqu'on compile le corps d'une fonction, et une liste vide sinon, lorsqu'on compile le programme principal. On notera qu'une variable qui n'est pas dans E est nécessairement l'argument de la fonction. Un aide-mémoire x86-64 est donné en annexe.

Question 14 Donner le code du compilateur pour
- la soustraction $c_1 - c_2$;
- la conditionnelle `if` c_1 `<= 0 then` c_2 `else` c_3 ;
- l'application $c_1\ c_2$.

Point fixe. Pour compiler la construction `fix` c, on choisit de représenter son résultat comme une fermeture de 16 octets, le premier mot contenant l'adresse du code d'une fonction assembleur `_fix` et le second la valeur de c, qui se trouve être une fermeture (c'est garanti par le typage).

Question 15 Justifier l'utilisation des instructions `pushq` et `popq` sur les registres `%rdi` et `%rsi` dans la compilation de la construction `fix` donnée figure 3.

Question 16 Donner le code assembleur de la fonction `_fix`.

Question 17 Quel est l'intérêt de représenter le résultat de `fix` c par une fermeture ?

4 Analyse syntaxique

On cherche maintenant à définir une syntaxe concrète pour notre langage, qui puisse être reconnue par un analyseur LR(1). Pour lever un certain nombre d'ambiguïtés telles que

```
print 1 - 2
```

on choisit de limiter les expressions en position d'argument d'une application, de `print` et de `fix` à des expressions réduites à des constantes entières, des identificateurs et des expressions parenthésées. On écrit donc notre grammaire sous la forme suivante :

$$
\begin{array}{rcl}
E & \to & S \\
 & | & E \texttt{ - } E \\
 & | & E\ S \\
 & | & \texttt{let ident = } E \texttt{ in } E \\
 & | & \texttt{fun ident -> } E \\
 & | & \texttt{if } E \texttt{ <= 0 then } E \texttt{ else } E \\
 & | & \texttt{fix } S \\
 & | & \texttt{print } S \\
S & \to & \texttt{constante} \\
 & | & \texttt{ident} \\
 & | & \texttt{(} E \texttt{)}
\end{array}
$$

Les non terminaux sont E et S. Tous les autres symboles sont terminaux. Les mots `let`, `in`, `fun`, `if`, `then`, `else`, `fix` et `print` sont des mots-clés.

Question 18 Une telle grammaire est cependant toujours refusée par l'outil `menhir` comme n'étant pas LR(1). Expliquer pourquoi.

Question 19 Proposer des règles de priorités et d'associativités qui permettent à `menhir` d'accepter cette grammaire (*i.e.* sans signaler de conflit).

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` `$`n`,` r_1	charge la constante n dans le registre r_1
`mov` `$`L`,` r_1	charge l'adresse de l'étiquette L dans le registre r_1
`sub` 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
`jg` L	saute à l'adresse désignée par l'étiquette L si les drapeaux signalent un résultat signé > 0
`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
`jmp` `*`o	saute à l'adresse désignée par l'opérande o
`call` `*`o	saute à l'adresse désignée par l'opérande o, 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`.