École Normale Supérieure

Langages de programmation et compilation

examen 2016–2017

Jean-Christophe Filliâtre

20 janvier 2017

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

---

1 Éloge de la simplicité

Dans ce problème, on considère la variante de mini-ML suivante, avec des entiers, une opération de soustraction et une construction `ifzero` qui permet de tester si une valeur est ou non l'entier 0.

e	::=	n	constante entière $n \in \mathbb{Z}$
	$\mid$	x	variable
	$\mid$	$e - e$	soustraction
	$\mid$	$\texttt{fun}\ x \to e$	fonction anonyme
	$\mid$	$e\ e$	application
	$\mid$	$\texttt{let}\ x = e\ \texttt{in}\ e$	variable locale
	$\mid$	$\texttt{ifzero}\ e\ \texttt{then}\ e\ \texttt{else}\ e$	conditionnelle

On munit ce langage d'une sémantique similaire à celle vue en cours, c'est-à-dire une stratégie d'appel par valeur avec évaluation de la gauche vers la droite. Les valeurs sont

v	::=	n	constante
	$\mid$	$\texttt{fun}\ x \to e$	fonction

Une sémantique opérationnelle à petits pas est donnée figure 1. On notera qu'elle est déterministe.

Comme on l'a vu en cours, il existe dans ce langage des expressions dont l'évaluation ne termine pas, comme $(\texttt{fun}\ x \to x\ x)\ (\texttt{fun}\ x \to x\ x)$. Mais on peut montrer en revanche que les expressions typées avec des *types simples* terminent toujours. C'est ce que nous allons faire dans les questions qui suivent. On se limite ici au seul type de base `int` et les types simples sont donc les suivants :

$$\tau \ ::= \ \texttt{int} \mid \tau \to \tau$$

Le jugement de typage est noté

$$\Gamma \vdash e : \tau$$

où l'environnement de typage Γ est une fonction des variables vers les types. Les règles d'inférence de ce jugement sont données figure 2. On admet les propriétés de progression (si $\vdash e : \tau$, alors soit e est une valeur, soit il existe e' tel que $e \to e'$) et de préservation (si $\vdash e : \tau$ et $e \to e'$ alors $\vdash e' : \tau$) pour ce langage.

contextes de réduction

$$
\begin{array}{rcl}
E & ::= & \square \\
& | & E\ e \\
& | & v\ E \\
& | & E - e \\
& | & v - E \\
& | & \texttt{let}\ x = E\ \texttt{in}\ e \\
& | & \texttt{ifzero}\ E\ \texttt{then}\ e\ \texttt{else}\ e
\end{array}
$$

réductions de tête

$$
\begin{array}{rcll}
(\texttt{fun}\ x \to e)\ v & \xrightarrow{\epsilon} & e[x \leftarrow v] & \\
\texttt{let}\ x = v\ \texttt{in}\ e & \xrightarrow{\epsilon} & e[x \leftarrow v] & \\
n_1 - n_2 & \xrightarrow{\epsilon} & n & \text{avec } n \text{ l'entier } n_1 - n_2 \\
\texttt{ifzero}\ 0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 & \xrightarrow{\epsilon} & e_1 & \\
\texttt{ifzero}\ n\ \texttt{then}\ e_1\ \texttt{else}\ e_2 & \xrightarrow{\epsilon} & e_2 & \text{si } n \neq 0
\end{array}
$$

FIGURE 1 – Sémantique opérationnelle à petits pas.

$$\frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)}$$

$$\frac{\Gamma + x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}\ x \to e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_1}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 - e_2 : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma + x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

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

FIGURE 2 – Règles de typage.

Pour un type τ donné, on définit un ensemble T_τ d'expressions closes de type τ par récurrence sur la structure de τ de la manière suivante :

1. $e \in T_{\texttt{int}}$ ssi l'évaluation de e termine ;
2. $e \in T_{\tau_1 \to \tau_2}$ ssi l'évaluation de e termine et quel que soit $e_1 \in T_{\tau_1}$, on a $e\ e_1 \in T_{\tau_2}$.

Notre objectif est de montrer que T_τ contient *toutes* les expressions closes de type τ.

Question 1 Montrer que si $\vdash e : \tau$ et $e \to e'$ alors $e \in T_\tau$ ssi $e' \in T_\tau$.

Correction : Par récurrence sur τ. Il est clair que l'évaluation de e termine si et seulement si celle de e' termine, car la sémantique est déterministe. Si $\tau = \texttt{int}$ c'est terminé. Sinon, $\tau = \tau_1 \to \tau_2$. ($\Rightarrow$) Supposons $e \in T_\tau$ et soit $e_1 \in T_{\tau_1}$. Par définition de T, on a $e\ e_1 \in T_{\tau_2}$. Or $e\ e_1 \to e'\ e_1$ et donc par HR sur τ_2 on a $e'\ e_1 \in T_{\tau_2}$. ($\Leftarrow$) Supposons $e' \in T_\tau$ et soit $e_1 \in T_{\tau_1}$. Par définition de T, on a $e'\ e_1 \in T_{\tau_2}$. Or $e\ e_1 \to e'\ e_1$ et donc par HR sur τ_2 on a $e\ e_1 \in T_{\tau_2}$.

Question 2 Soit e telle que $x_1 : \tau_1 + \cdots + x_k : \tau_k \vdash e : \tau$ et $v_1, \ldots, v_k$ des valeurs closes de types respectifs $\tau_1, \ldots, \tau_k$ telles que $v_i \in T_{\tau_i}$ pour tout i. Montrer alors que

$$e[x_1 \leftarrow v_1] \cdots [x_k \leftarrow v_k] \in T_\tau.$$

Correction : On note Γ l'environnement $x_1 : \tau_1 + \cdots + x_k : \tau_k$ et σ la substitution $[x_1 \leftarrow v_1] \cdots [x_k \leftarrow v_k]$. On procède par récurrence sur la dérivation de typage de e.

$x = n$ **:** Immédiat par définition de $T_{\texttt{int}}$.

$e = x$ **:** Immédiat, car x est nécessairement l'un des x_i et donc $e\sigma = v_i$ qui appartient à T_{τ_i} par hypothèse.

$e = e_1 - e_2$ **:** e_1 et e_2 sont bien typées de type $\texttt{int}$ et on peut donc leur appliquer l'HR. On a donc $e_1\sigma \in T_{\texttt{int}}$ et $e_2\sigma \in T_{\texttt{int}}$, c'est-à-dire que l'évaluation de $e_1\sigma$ et de $e_2\sigma$ terminent. On doit montrer $e_1\sigma - e_2\sigma \in T_{\texttt{int}}$, c'est-à-dire que l'évaluation de $e_1\sigma - e_2\sigma$ termine, ce qui est clair.

$e = \texttt{fun}\ x \to e_1$ **:** C'est là le cas compliqué. On a $\Gamma + x : \tau' \vdash e_1 : \tau''$ avec $\tau = \tau' \to \tau''$. L'évaluation de $e\sigma$ termine clairement car $e\sigma = \texttt{fun}\ x \to e_1\sigma$ donc une valeur. Soit $e' \in T_{\tau'}$. Montrons que $e\sigma\ e' \in T_{\tau''}$. Par définition de T, on sait que l'évaluation de e' termine ; soit v sa valeur. Par la question précédente, on a $v \in T_{\tau'}$. Par HR sur e_1, on a donc que $e_1\sigma[x \leftarrow v] \in T_{\tau''}$. Or

$$\begin{array}{rl} & e\sigma\ e' \\ \to^\star & e\sigma\ v \\ = & \texttt{fun}\ x \to e_1\sigma\ v \\ \to & e_1\sigma[x \leftarrow v] \end{array}$$

Par application de la question précédente, on en déduit $e\sigma\ e' \in T_{\tau''}$. Et donc finalement $e\sigma \in T_\tau$ par définition de T.

$e = e_1\ e_2$ **:** On a $\Gamma \vdash e_1 : \tau' \to \tau$ et $\Gamma \vdash e_2 : \tau'$. Par HR on a $e_1\sigma \in T_{\tau' \to \tau}$ et $e_2\sigma \in T_{\tau'}$. Par définition de $T_{\tau' \to \tau}$, on a $e_1\sigma\ e_2\sigma \in T_\tau$, c'est-à-dire $(e_1\ e_2)\sigma \in T_\tau$.

$e = \texttt{let}\ x = e_1\ \texttt{in}\ e_2$ **:** On a $\Gamma \vdash e_1 : \tau'$ et $\Gamma + x : \tau' \vdash e_2 : \tau$. Par HR on a $e_1\sigma \in T_{\tau'}$. Donc l'évaluation de $e_1\sigma$ termine ; soit v sa valeur. On a

$$\begin{array}{rl} & (\texttt{let}\ x = e_1\ \texttt{in}\ e_2)\sigma \\ = & \texttt{let}\ x = e_1\sigma\ \texttt{in}\ e_2\sigma \\ \to^\star & \texttt{let}\ x = v\ \texttt{in}\ e_2\sigma \\ \to & e_2\sigma[x \leftarrow v] \end{array}$$

et par HR sur e_2 on a $e_2\sigma[x \leftarrow v] \in T_\tau$. Par application de la question précédente, on en déduit $(\texttt{let}\ x = e_1\ \texttt{in}\ e_2)\sigma \in T_\tau$.

$e = \texttt{ifzero}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$ **:** On a $\Gamma \vdash e_1 : \texttt{int}$, $\Gamma \vdash e_2 : \tau$ et $\Gamma \vdash e_3 : \tau$. Par HR on a $e_1\sigma \in T_{\texttt{int}}$. Donc l'évaluation de $e_1\sigma$ termine ; soit n sa valeur. On a

$$\begin{array}{rl} & (\texttt{ifzero}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3)\sigma \\ = & \texttt{ifzero}\ e_1\sigma\ \texttt{then}\ e_2\sigma\ \texttt{else}\ e_3\sigma \\ \to^\star & \texttt{ifzero}\ n\ \texttt{then}\ e_2\sigma\ \texttt{else}\ e_3\sigma \end{array}$$

Si $n = 0$ une réduction de plus donne $e_2\sigma$ qui appartient à T_τ par HR. Par application de la question précédente, on en déduit $e\sigma \in T_\tau$. De même si $n \neq 0$ avec e_3 au lieu de e_2.

Question 3 En déduire que l'évaluation de toute expression close typée termine.

Correction : Si e est une expression close telle que $\vdash e : \tau$ alors $e \in T_\tau$ par la question précédente en prenant $k = 0$. Et donc l'évaluation de e termine par définition de T_τ.

Variables mutables. On étend maintenant notre langage avec la possibilité de modifier en place la valeur d'une variable. On ajoute au langage une nouvelle construction pour cela :

$$\begin{array}{rcll} e & ::= & \ldots & \\ & | & x \leftarrow e & \text{affectation} \end{array}$$

On suppose de plus que l'on distingue les variables introduites par `let` et les variables introduites par `fun` et que seules les variables introduites par `let` peuvent apparaître à gauche de l'affectation.

Pour donner une sémantique à ce nouveau langage, on introduit une notion d'adresse, notée a. On ne précise pas ce que sont les adresses mais on les suppose en nombre infini. Un état mémoire, noté M, est une fonction des adresses vers les valeurs. La notion de valeur ne change pas. En particulier, les adresses ne sont pas des valeurs. Pour définir une sémantique opérationnelle à petits pas sur ce nouveau langage, on ajoute les adresses à la syntaxe abstraite des expressions :

$$\begin{array}{rcll} e & ::= & \ldots & \\ & | & a & \text{accès à une adresse} \\ & | & a \leftarrow e & \text{affectation à une adresse} \end{array}$$

De telles adresses ne font pas partie du langage dans lequel l'utilisateur écrit des programmes, mais sont introduites uniquement pour les besoins de la sémantique opérationnelle. La relation de réduction en un pas prend la forme

$$M_1/e_1 \rightsquigarrow M_2/e_2$$

$$
\begin{array}{rcll}
M/a & \overset{\epsilon}{\rightsquigarrow} & M/M(a) & \\
M/(\texttt{fun}\ x \to e)\ v & \overset{\epsilon}{\rightsquigarrow} & M/e[x \leftarrow v] & \\
M/\texttt{let}\ x = v\ \texttt{in}\ e & \overset{\epsilon}{\rightsquigarrow} & M[a \mapsto v]/e[x \leftarrow a] & \text{avec } a \text{ une adresse fraîche} \\
M/a \leftarrow v & \overset{\epsilon}{\rightsquigarrow} & M[a \mapsto v]/v & \\
M/n_1 - n_2 & \overset{\epsilon}{\rightsquigarrow} & M/n & \text{avec } n \text{ l'entier } n_1 - n_2 \\
M/\texttt{ifzero}\ 0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 & \overset{\epsilon}{\rightsquigarrow} & M/e_1 & \\
M/\texttt{ifzero}\ n\ \texttt{then}\ e_1\ \texttt{else}\ e_2 & \overset{\epsilon}{\rightsquigarrow} & M/e_2 & \text{si } n \neq 0
\end{array}
$$

FIGURE 3 – Sémantique opérationnelle avec état mémoire.

et s'interprète comme « l'état mémoire M_1 et l'expression e_1 se réduisent en un pas de calcul vers l'état mémoire M_2 et l'expression e_2 ». Les contextes de réduction traduisent toujours une stratégie d'appel par valeur avec évaluation de la gauche vers la droite. Ce sont donc les mêmes qu'à la figure 1, avec l'addition d'un nouveau contexte pour réduire à droite d'une affectation :

$$
\begin{array}{rcl}
E & ::= & \ldots \\
 & | & a \leftarrow E
\end{array}
\qquad\qquad
\frac{M/e \overset{\epsilon}{\rightsquigarrow} M'/e'}{M/E(e) \rightsquigarrow M'/E(e')}
$$

Les réductions de tête sont données figure 3.

Question 4 Donner la séquence de réductions de l'expression

$$
\begin{array}{l}
\texttt{let}\ x = 1\ \texttt{in} \\
\texttt{let}\ f = \texttt{fun}\ y \to x \leftarrow x - y\ \texttt{in} \\
(x \leftarrow 0) - (f\ 42)
\end{array}
$$

dans un état mémoire initialement vide.

Correction :

$$
\begin{array}{rl}
 & / e \\
\rightsquigarrow & a_1 \mapsto 1\ /\ \texttt{let}\ f = \texttt{fun}\ y \to a_1 \leftarrow a_1 - y\ \texttt{in}\ (a_1 \leftarrow 0) - (f\ 42) \\
\rightsquigarrow & a_1 \mapsto 1 + a_2 \mapsto \texttt{fun}\ y \to a_1 \leftarrow a_1 - y\ /\ (a_1 \leftarrow 0) - (a_2\ 42) \\
\rightsquigarrow & a_1 \mapsto 0 + a_2 \mapsto \texttt{fun}\ y \to a_1 \leftarrow a_1 - y\ /\ 0 - (a_2\ 42) \\
\rightsquigarrow & \ldots\ /\ 0 - ((\texttt{fun}\ y \to a_1 \leftarrow a_1 - y)\ 42) \\
\rightsquigarrow & \ldots\ /\ 0 - (a_1 \leftarrow a_1 - 42) \\
\rightsquigarrow & \ldots\ /\ 0 - (a_1 \leftarrow 0 - 42) \\
\rightsquigarrow & \ldots\ /\ 0 - (a_1 \leftarrow -42) \\
\rightsquigarrow & a_1 \mapsto -42 + \ldots\ /\ 0 - (-42) \\
\rightsquigarrow & a_1 \mapsto -42 + a_2 \mapsto \texttt{fun}\ y \to a_1 \leftarrow a_1 - y\ /\ 42
\end{array}
$$

Typage. On type ce nouveau langage avec les mêmes types simples que précédemment. L'environnement de typage Γ, qui donne le type des variables, reste le même. Pour typer les adresses, on se donne un second environnement, à savoir une fonction Σ des adresses vers les types. Le jugement de typage prend la forme $\Gamma, \Sigma \vdash e : \tau$. Les règles de typage de la figure 2 sont inchangées, si ce n'est que Σ est ajouté à côté de tout environnement.

Question 5 Donner les règles de typage pour les nouvelles constructions, à savoir une adresse a, une affectation $x \leftarrow e$ et une affectation $a \leftarrow e$.

Correction : Pas de difficulté pour une adresse :

$$\frac{a \in \mathrm{dom}(\Sigma)}{\Gamma, \Sigma \vdash a : \Sigma(a)}$$

Pour l'affectation, on donne à l'expression toute entière le type de la valeur affectée pour être cohérent avec la règle de sémantique :

$$\frac{\Gamma, \Sigma \vdash e : \tau \quad \Gamma(x) = \tau}{\Gamma, \Sigma \vdash x \leftarrow e : \tau} \qquad \frac{\Gamma, \Sigma \vdash e : \tau \quad \Sigma(a) = \tau}{\Gamma, \Sigma \vdash a \leftarrow e : \tau}$$

Correction du typage. Pour adapter la preuve de correction du typage vue en cours, il faut mettre en relation les états mémoire et les environnements. On dit qu'un état mémoire M est bien typé dans un environnement Γ, Σ, ce que l'on note $\Gamma, \Sigma \vdash M$, si $\mathrm{dom}(M) = \mathrm{dom}(\Sigma)$ et $\Gamma, \Sigma \vdash M(a) : \Sigma(a)$ pour tout $a \in \mathrm{dom}(M)$.

Question 6 Montrer la préservation du typage par la réduction, c'est-à-dire que, si $\Gamma, \Sigma \vdash e : \tau$, $\Gamma, \Sigma \vdash M$ et $M/e \rightsquigarrow M'/e'$, alors il existe un environnement Σ' tel que $\Gamma, \Sigma' \vdash e' : \tau$ et $\Gamma, \Sigma' \vdash M'$. (On admet les résultats de permutation, d'affaiblissement et de préservation par substitution analogues à ceux vus en cours.)

Correction : Par récurrence sur la dérivation de typage.

$e = \texttt{let}\ x = e_1\ \texttt{in}\ e_2$ **:** deux réductions possibles
- si $M/e_1 \rightsquigarrow M'/e_1'$ alors par HR il existe Σ' tel que $\Gamma + x : \tau_1, \Sigma' \vdash e_1' : \tau_1$ et $\Gamma + x : \tau_1, \Sigma' \vdash M'$, et donc $\Gamma, \Sigma' \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2$ et $\Gamma\Sigma' \vdash M'$.
- si en revanche e_1 est une valeur v, alors la réduction est $M/\texttt{let}\ x = v\ \texttt{in}\ e_2 \rightsquigarrow M[a \mapsto v]/e_2[x \leftarrow a]$. C'est là le cas intéressant, où les choses se passent. On prend $M' = M[a \mapsto v]$ et $\Sigma' = \Sigma + a : \tau_1$. Par le lemme de préservation par substitution (admis), $e_2[x \leftarrow a]$ est toujours de type τ. Et par ailleurs $\Gamma, \Sigma' \vdash M'$ car on a choisi $\Sigma'(a) = \tau_1$.

$e = a \leftarrow e_1$ **:** deux réductions possibles
- si e_1 se réduit on applique l'HR.
- si e_1 est une valeur v, alors la réduction est $M/a \leftarrow v \rightsquigarrow M[a \mapsto v]/v$. On pose alors $M' = M[a \mapsto v]$ et $\Sigma' = \Sigma$. Le réduit v est de type τ donc le type est préservé. Par ailleurs $\tau = \Sigma(a)$ donc $\Gamma, \Sigma \vdash M'$.

$e = e_1\ e_2$ **:** trois réductions possibles
- si e_1 ou e_2 se réduit, on applique l'HR.
- si $e_1 = \texttt{fun}\ x \to e$ et $e_2 = v$ alors $e' = e[x \leftarrow v]$ et on applique le lemme de substitution, en prenant $M' = M$ et $\Sigma' = \Sigma$.

$e = a$ **:** la réduction est $M/a \rightsquigarrow M/M(a)$. On a $\tau = \Sigma(a)$. Comme $\Gamma, \Sigma \vdash M$, on sait que $M(a)$ a le type $\Sigma(a)$, c'est-à-dire τ. On pose $M' = M$ et $\Sigma' = \Sigma$ pour conclure.

cas $e_1 - e_2$ **et** `ifzero` **:** pour les réductions de tête, on pose $M' = M$ et $\Sigma' = \Sigma$ et le résultat est évident. Pour les réductions dans un contexte, on applique l'HR au sous-terme qui se réduit.

Question 7 Montrer la propriété de progression, c'est-à-dire que si $\emptyset, \Sigma \vdash e : \tau$ alors soit e est une valeur, soit pour tout M tel que $\emptyset, \Sigma \vdash M$ il existe M' et e' tels que $M/e \rightsquigarrow M', e'$.

Correction : Par récurrence sur la dérivation de typage.

$e = n$ **ou** $e = \texttt{fun}\ x \to e_1$ **:** immédiat car e est une valeur.

$e = a$ **:** comme e est bien typée, on a $a \in \mathrm{dom}(M)$ et donc $M/a \rightsquigarrow M/M(a)$.

$e = \texttt{let}\ x = e_1\ \texttt{in}\ e_2$ **:** par HR, e_1 est une valeur ou se réduit.
- si $M/e_1 \rightsquigarrow M'/e_1'$, alors $M/\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \rightsquigarrow M'/\texttt{let}\ x = e_1'\ \texttt{in}\ e_2$ (passage au contexte).
- sinon, e_1 est une valeur v et $M/\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \rightsquigarrow M[a \mapsto v]/e[x \leftarrow a]$. Il suffit de prendre $M' = M[a \mapsto v]$.

$e = e_1\ e_2$ **:** par HR, e_1 est une valeur ou se réduit.
- si $M/e_1 \rightsquigarrow M'/e_1'$, alors $M/e_1\ e_2 \rightsquigarrow M'/e_1'\ e_2$ (passage au contexte).
- sinon, e_1 est une valeur et c'est nécessairement une valeur de la forme $\texttt{fun}\ x \to e$ (typage). Par HR, e_1 est une valeur ou se réduit.
  - si $M/e_2 \rightsquigarrow M'/e_2'$, alors $M/e_1\ e_2 \rightsquigarrow M'/e_1\ e_2'$ (passage au contexte).
  - sinon, e_2 est une valeur v et $M/e_1\ e_2 \rightsquigarrow M/e[x \leftarrow v]$.

$e = a \leftarrow e_1$ **:** par HR, e_1 est une valeur ou se réduit.
- si $M/e_1 \rightsquigarrow M'/e_1'$, alors $M/a \leftarrow e_1 \rightsquigarrow M'/a \leftarrow e_1'$ (passage au contexte).
- sinon, e_1 est une valeur v et on a $M/a \leftarrow v \rightsquigarrow M[a \mapsto v]/v$. Il suffit de prendre $M' = M[a \mapsto v]$.

autres cas : se traitent de façon similaire, en appliquant l'HR aux sous-expressions. Comme ci-dessus, on a à chaque fois une réduction par passage ou contexte ou une réduction en tête. Dans ce dernier cas, le typage intervient de façon cruciale pour assurer que les valeurs soustraites sont bien des entiers ou que la valeur testée par `ifzero` est bien un entier.

Question 8 En déduire la sûreté du typage, à savoir que si $\emptyset, \Sigma \vdash e : \tau$, si $\emptyset, \Sigma \vdash M$ et si $M/e \rightsquigarrow^\star M', e'$ avec M'/e' irréductible, alors e' est une valeur.

Correction : On a

$$M/e \rightsquigarrow M_1/e_1 \rightsquigarrow \cdots \rightsquigarrow M'/e'$$

et par applications répétées du résultat de préservation, on a donc $\emptyset, \Sigma' \vdash e' : \tau$ pour un certain Σ'. Par la propriété de progrès, e' est donc une valeur.

Question 9 Donner une expression close e de type $\texttt{int} \to \texttt{int}$ correspondant à la fonction factorielle sur les entiers naturels, c'est-à-dire telle que $e\ n$ s'évalue en $n!$ pour tout $n \in \mathbb{N}$.

Correction : Pour écrire une fonction récursive, on va utiliser une variable contenant une fonction, initialisée avec une fonction arbitraire, puis modifiée par une fonction s'appelant

elle-même. C'est ce qu'on appelle le nœud de Landin. On le fait deux fois, une fois pour écrire une fonction de multiplication et une fois pour la fonction factorielle.

$$\begin{array}{l}\texttt{let } \mathit{mult} = \texttt{fun } x \to \texttt{fun } y \to 0 \texttt{ in}\\ \texttt{let \_} = \mathit{mult} \leftarrow (\texttt{fun } x \to \texttt{fun } y \to \texttt{ifzero } x \texttt{ then } 0 \texttt{ else } (\mathit{mult}\ (x-1)\ y) - (0-y)) \texttt{ in}\\ \texttt{let } \mathit{fact} = \texttt{fun } x \to 0 \texttt{ in}\\ \texttt{let \_} = \mathit{fact} \leftarrow (\texttt{fun } x \to \texttt{ifzero } x \texttt{ then } 1 \texttt{ else } \mathit{mult}\ x\ (\mathit{fact}\ (x-1))) \texttt{ in}\\ \mathit{fact}\end{array}$$

Question 10 Donner une expression close e de type `int` dont l'évaluation ne termine pas.

Correction : On réutilise la même idée pour faire une boucle infinie :

$$\begin{array}{l}\texttt{let } f = \texttt{fun } x \to 0 \texttt{ in}\\ \texttt{let \_} = f \leftarrow (\texttt{fun } x \to f\ x) \texttt{ in}\\ f\ 42\end{array}$$

Remarque : on aurait pu tout aussi bien donner $\mathit{fact}\ (-1)$ avec la fonction *fact* précédente.

Question 11 Proposer un schéma de compilation pour ce langage. On discutera notamment de la possibilité d'allouer certaines variables sur la pile.

Correction : Il n'est pas facile d'allouer les variables sur la pile, car elles peuvent être capturées dans des fermetures. Par ailleurs, elles doivent être capturées *par référence* pour respecter la sémantique, c'est-à-dire que dans une expression comme

$$\begin{array}{l}\texttt{let } x = 0 \texttt{ in}\\ \texttt{let } f = \texttt{fun } y \to \ldots x \leftarrow e \ldots \texttt{ in}\\ \ldots\end{array}$$

la fonction f modifie la variable x introduite par le premier `let` et non pas une autre variable qui aurait été introduite au moment de la construction de la fermeture.

Si on renonce à l'idée d'allouer certaines variables sur la pile, le schéma de compilation le plus simple consiste à compiler ce langage exactement comme on compilerait OCaml avec la traduction T définie par

$$\begin{array}{rcl} T(\texttt{let } x = e_1 \texttt{ in } e_2) & = & \texttt{let } x = \texttt{ref } T(e_1) \texttt{ in } T(e_2)\\ T(x) & = & \texttt{!}x \quad \text{pour une variable mutable}\\ T(x) & = & x \quad \text{sinon}\\ T(x \leftarrow e) & = & x \texttt{ := } T(e)\end{array}$$

et un morphisme pour les autres constructions.

Pour allouer certaines variables sur la pile, il faudrait faire une *analyse d'échappement* pour déterminer les variables mutables qui ne sortent pas de la portée de la fonction courante, c'est-à-dire ne sont pas capturées par des fonctions qui sont renvoyées ou stockées par effet de bord dans des variables extérieures à la fonction courante.

Capture par valeur. On souhaite ajouter à notre langage la possibilité de capturer une variable *par valeur* à l'intérieur d'une fermeture. Pour cela, on modifie la syntaxe des fonctions de la manière suivante :

$$\begin{array}{rcll} e & ::= & \ldots & \\ & | & \texttt{fun}\ x\ \{x_1, \ldots, x_n\} \to e & \text{fonction} \end{array}$$

L'ensemble $\{x_1, \ldots, x_n\}$ représente le sous-ensemble des variables libres de e que l'on souhaite capturer par valeur. Lorsque cet ensemble est vide, la construction coïncide avec la construction $\texttt{fun}\ x \to e$ précédente. Une variable capturée par valeur reste une variable mutable à l'intérieur du corps de la fonction ; mais elle représente une *nouvelle* variable. Ainsi, les deux expressions suivantes

$$\begin{array}{ll} \texttt{let}\ x = 0\ \texttt{in} & \texttt{let}\ x = 0\ \texttt{in} \\ \texttt{let}\ f = \texttt{fun}\ y \to x \leftarrow y - x\ \texttt{in} & \texttt{let}\ f = \texttt{fun}\ y\ \{x\} \to x \leftarrow y - x\ \texttt{in} \\ (f\ 42) - x & (f\ 42) - x \end{array}$$

s'évaluent respectivement en 0 et 42.

Question 12 Proposer une traduction de ce troisième langage vers le précédent, c'est-à-dire une élimination de la construction $\texttt{fun}\ x\ \{x_1, \ldots, x_n\} \to e$ au profit de la construction $\texttt{fun}\ x \to e$ précédente, qui respecte la sémantique.

Correction : Soit T cette transformation. Il suffit de prendre

$$\begin{array}{rcl} T(\texttt{fun}\ x\ \{x_1, \ldots, x_n\} \to e) & = & \texttt{let}\ x_1 = x_1\ \texttt{in} \\ & & \ldots \\ & & \texttt{let}\ x_n = x_n\ \texttt{in} \\ & & \texttt{fun}\ x \to T(e) \end{array}$$

On capture ainsi les valeurs des variables $x_1, \ldots, x_n$ dans de *nouvelles* variables (de mêmes noms), qui restent des variables mutables. La transformation T est un morphisme pour toutes les autres constructions.

Question 13 En supposant que *toutes* les variables sont capturées par valeur, montrer que de nouveau l'évaluation de toute expression close bien typée termine. (Attention : ce n'est pas le même langage qu'au départ, puisque les variables restent mutables.)

Correction : Il suffit de remplacer toute expression

$$x \leftarrow e$$

par l'expression

$$\texttt{let}\ x = e\ \texttt{in}$$

pour préserver la sémantique et le typage. On se retrouve donc maintenant dans le cadre du premier langage, pour lequel on a montré la terminaison des programmes bien typés.

---

2 Un visiteur, venu d'ailleurs [1]

Dans ce problème, on s'intéresse à l'écriture d'un interprète dans un langage orienté objet, en l'occurrence Java. Prenons l'exemple minimaliste d'un langage d'expressions réduites à des constantes entières et des soustractions, c'est-à-dire la syntaxe abstraite suivante :

$$\begin{array}{rcll} e & ::= & n & \text{constante entière } n \in \mathbb{Z} \\ & | & e - e & \text{soustraction} \end{array}$$

Une telle syntaxe abstraite peut être représentée en Java par trois classes, dont on omet ici les constructeurs :

```
abstract class Expr { }
class Ecte extends Expr { int n; }
class Esub extends Expr { Expr e1, e2; }
```

La classe `Ecte` représente une constante entière n et la classe `Esub` une expression de la forme $e_1 - e_2$. On peut écrire un interprète pour ces expressions en introduisant une méthode `interp` dans la classe `Expr`, de la manière suivante

```
abstract class Expr { abstract int interp(); }
```

et en la définissant dans chacune des deux sous-classes.

Question 14 Donner le code Java de la méthode `interp` pour chacune des deux sous-classes `Ecte` et `Esub`.

Correction :

```
class Ecte extends Expr { ...
  int interp() { return this.n; } }
class Esub extends Expr { ...
  int interp() { return this.e1.interp() - this.e2.interp(); } }
```

Une autre solution. Bien que parfaitement correcte, cette façon de procéder a le défaut d'être intrusive, car elle mélange les données (c'est-à-dire ici les types, champs et constructeurs définissant la syntaxe abstraite) et l'algorithme (c'est-à-dire ici le code de l'interprète). Il existe une autre façon de procéder, connue sous le nom de *visiteur*, permettant de séparer données et algorithme. L'idée est d'écrire le code de l'interprète dans une classe à part `Interp`

```
class Interp {
  int visit(Ecte e) { ... }
  int visit(Esub e) { ... }
}
```

et de se contenter d'une intrusion minimale dans la syntaxe abstraite sous la forme d'une méthode « acceptant » le visiteur.

```
abstract class Expr { abstract int accept(Interp i); }
class Ecte extends Expr { ...; int accept(Interp i) { return i.visit(this); }
class Esub extends Expr { ...; int accept(Interp i) { return i.visit(this); }
```

1. Hommage à *Toy Story*, évidemment.

Question 15 Donner le code Java de la classe `Interp`.

Correction :

```
class Interp {
  int visit(Ecte e) { return e.n; }
  int visit(Esub e) { return e.e1.accept(this) - e.e2.accept(this); }
}
```

Question 16 Le code de la méthode `accept` étant rigoureusement le même dans les classes `Ecte` et `Esub`, on peut être tenté de l'écrire dans la classe `Expr`, afin que les classes `Ecte` et `Esub` en héritent. (Java permet en effet de définir une méthode dans une classe abstraite, exactement pour cette raison.) Expliquer pourquoi ce n'est pas possible ici.

Correction : Dans les classes `Ecte` et `Esub`, `this` n'a pas le même type. Du coup, quand on écrit `i.visit(this)` dans la classe `Ecte` on appelle la méthode `visit(Ecte e)` de l'interprète, alors qu'on appelle la méthode `visit(Esub e)` quand on est dans la classe `Esub`.

On le verrait plus simplement encore si on n'avait pas utilisé ici la surcharge de Java et écrit plutôt

```
class Interp {
  int visitCte(Ecte e) { ... }
  int visitSub(Esub e) { ... }
}
```

et donc

```
class Ecte extends Expr { ...
  int accept(Interp i) { return i.visitCte(this); }
class Esub extends Expr { ...
  int accept(Interp i) { return i.visitSub(this); }
```

On voit bien alors que le code n'est *pas* le même.

Question 17 Donner le code assembleur x86-64 de la méthode `accept` de la classe `Ecte` ainsi que des deux méthodes `visit` de la classe `Interp`. On suppose un schéma de compilation analogue à celui du cours : les objets sont des pointeurs vers le tas et leur premier champ contient un pointeur vers le descripteur de classe ; pour une méthode, l'objet `this` est passé dans `%rdi` et l'argument dans `%rsi` (il n'y a qu'un argument ici à chaque fois).

Correction :

```
Ecte_accept:
  mov  %rdi, %rcx     # échanger this (%rdi) et i (%rsi)
  mov  %rsi, %rdi
  mov  %rcx, %rsi
```

```
  mov  (%rdi), %rcx   # accès au descripteur de classe de i
  jmp  *8(%rcx)       # appel terminal à la méthode visit(Ecte)
                      # en la supposant rangée là dans le descripteur
Interp_visit_Ecte:
  mov  8(%rsi), %rax  # n = premier champ
  ret

Interp_visit_Esub:
  push %rdi           # sauvegarde this
  push %rsi           # sauvegarde e
  mov  8(%rsi), %rcx  # e1 = premier champ
  mov  %rdi, %rsi
  mov  %rcx, %rdi
  mov  (%rdi), %rcx   # descripteur de e.e1
  call *8(%rcx)       # e.e1.accept(this)
  pop  %rdi           # on restaure e et this
  pop  %rsi
  push %rax           # et on sauvegarde le résultat
  mov  16(%rdi), %rdi # e2 = second champ
  mov  (%rdi), %rcx   # descripteur de e.e2
  call *8(%rcx)       # e.e2.accept(this)
  mov  %rax, %rcx
  pop  %rax
  sub  %rcx, %rax
  ret
```

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 la somme de r_1 et r_2 dans r_1 (on a de même `sub` et `imul`)
`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
`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`.