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.

Correction : Choisissons par exemple d'évaluer de la gauche vers la droite :

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.

```
       (print 2) - ((print 60) - (print 100))
-  2-> 2         - ((print 60) - (print 100))
- 60-> 2         - (60         - (print 100))
-100-> 2         - (60         - 100        )
-----> 2         - (-40)
-----> 42
```

Il y a 8 évaluations possibles pour cette expression.
— si on commence par `print` 2, on a le choix ensuite entre `print` 60 ou `print` 100 et ensuite il n'y a plus de choix possible (2 évaluations) ;
— si on commence en revanche par `print` 60, on a le choix ensuite entre `print` 2 et plus de choix ensuite, ou `print` 100 et ensuite le choix entre $60 - 100$ ou `print` 2 (3 évaluations en tout) ;
— si on commence enfin par `print` 100, c'est analogue au cas précédent en échangeant le rôle de 60 et 100 (3 évaluations en tout).

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.

Correction :

```
let mult = fun n -> fix (fun mult -> fun m ->
                            if m <= 0 then 0 else n - (0 - mult (m - 1))) in
let fact = fix (fun fact -> fun n ->
                  if n <= 0 then 1 else mult n (fact (n - 1))) in
fun n -> print (fact 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 $e_r \to e'$.

Correction : On procède par récurrence sur e. Supposons $e_l \neq e_r$.
— Si la réduction est de tête, alors nécessairement $e_l = e_r$ (une seule réduction possible).
— Si la réduction est contextuelle, il y a deux cas :
  — les cas `print`, `if` et `fix` : on applique l'IR et on repasse au contexte les deux réductions obtenues ;
  — les cas d'une soustraction et d'une application :
    — si les deux réductions se font avec le même contexte, c'est identique au cas précédent ;
    — sinon, supposons par exemple $e = E_1(e_1) - e_2 \to E_1(e_1') - e_2$ et $e = e_1 - E_2(e_2) \to e_1' - E_2(e_2')$. Alors dans ce cas les deux termes se réduisent en une étape vers $e' = E_1(e_1') - E_2(e_2')$.

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 $m \leq n + n'$.

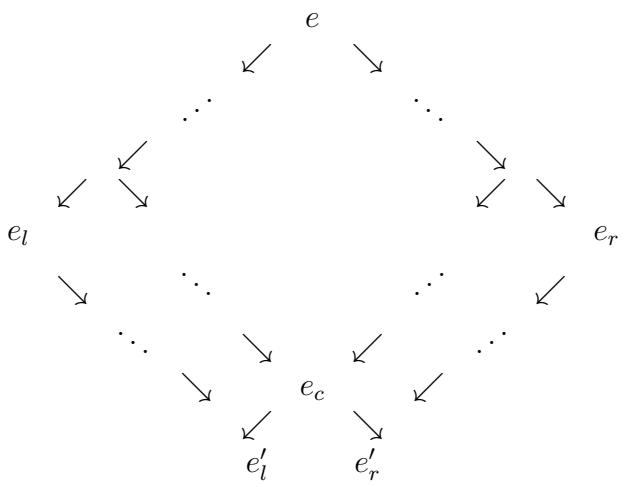
Correction : Par récurrence sur $n + n'$.

— si $n = 0$ alors la première réduction est étendue en la seconde, et la seconde est laissée inchangée (et on a $m = n'$) ;

— si $n > 0$,

  — le cas $n' = 0$ est identique au précédent

  — si $n' > 0$ on a une situation de la forme



Par IR sur $(n-1, n'-1)$ on peut clore les réductions en un terme e_c, avec le même nombre $m_c \leq n + n' - 2$ de réductions de chaque côté.
On peut ensuite clore le diagramme de chaque côté avec encore une IR, respectivement sur $(1, m_c - n - 1)$ et $(m_c - n' - 1, 1)$, vers deux termes e'_l et e'_r. S'ils sont égaux, on a terminé en au plus $n + n' - 1$ étapes. Sinon, on applique la question précédente, ce qui permet de clore le diagramme en au plus $n + n'$ étapes.

---

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 :

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

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

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

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

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

FIGURE 2 – Typage avec effets.

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

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

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.

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

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

Correction :

```
e1 : (print 1) - (print 2)
e2 : let tmp = print 0 in fun x -> print x
e3 : fun f -> print 0
```

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.

Correction : Pour typer cette expression, il faut donner à la variable f un type de la forme

$$(\tau_1 \xrightarrow{\phi_1} \tau_2) \xrightarrow{\phi_2} \tau_3$$

Or la première application de f exige que $\phi_1 = \bot$ car $\vdash \texttt{fun}\ x \to 0 : \tau_1 \xrightarrow{\bot} \texttt{int}\ \&\ \bot$. Et la seconde application de f exige que $\phi_1 = \texttt{P}$ car $\vdash \texttt{fun}\ x \to \texttt{print}\ 0 : \tau_1 \xrightarrow{\texttt{P}} \texttt{int}\ \&\ \bot$. Cette expression n'est donc 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 $e \xrightarrow{a} e'$.

Correction : Par récurrence sur la dérivation $\vdash e : \tau\ \&\ \phi$ et par cas sur la dernière règle :
- e ne peut être une variable.
- Si e est une constante ou une fonction, c'est une valeur.
- $e = e_1 - e_2$:
  - si e_1 n'est pas une valeur, alors par HR e_1 se réduit et par passage au contexte e se réduit également ;
  - de même si e_2 n'est pas une valeur ;
  - si enfin e_1 et e_2 sont des valeurs, alors ce sont des entiers (par typage) et donc $e_1 - e_2$ se réduit.
- $e = e_1\ e_2$: similaire à $e_1 - e_2$
- $e = \texttt{print}\ e_1$:
  - si e_1 n'est pas une valeur, alors par HR e_1 se réduit et par passage au contexte e se réduit ;
  - sinon, e_1 est un entier n (par typage) et donc $e = \texttt{print}\ n \xrightarrow{n} n$.
- $e = \texttt{if}\ e_1\ \texttt{<= 0 then}\ e_2\ \texttt{else}\ e_3$: similaire à `print` (soit e_1 se réduit, soit e_1 est un entier et `if` se réduit).
- $e = \texttt{fix}\ e_1$: similaire à `print` (soit e_1 se réduit, soit e_1 est une fonction et `fix` se réduit).

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 : Par récurrence sur la dérivation $\Gamma \vdash e : \tau\ \&\ \phi$ et par cas sur la dernière règle.
- e ne peut être une variable, ni une constante, ni une fonction (car e se réduit).
- $e = e_1 - e_2$: on a

$$\frac{\Gamma \vdash e_1 : \texttt{int}\ \&\ \phi_1 \quad \Gamma \vdash e_2 : \texttt{int}\ \&\ \phi_2}{\Gamma \vdash e_1 - e_2 : \texttt{int}\ \&\ \phi_1 \sqcup \phi_2}$$

— si c'est e_1 qui se réduit vers e'_1, alors par IR $\Gamma \vdash e'_1 : \texttt{int} \ \& \ \phi'_1$ avec $\phi'_1 \leq \phi_1$. Dès lors

$$\frac{\Gamma \vdash e'_1 : \texttt{int} \ \& \ \phi'_1 \quad \Gamma \vdash e_2 : \texttt{int} \ \& \ \phi_2}{\Gamma \vdash e'_1 - e_2 : \texttt{int} \ \& \ \phi'_1 \sqcup \phi_2}$$

et donc le réduit $e'_1 - e_2$ est bien typé, d'un effet inférieur ou égal (car $\sqcup$ est monotone).

— de même si c'est e_2 qui se réduit.

— si enfin e_1 et e_2 sont des entiers, alors $e_1 - e_2$ se réduit vers un entier, toujours bien typé (de même effet $\bot$ dans ce cas).

— $e = e_1 \ e_2$: comme pour la soustraction, en utilisant la propriété de substitution pour justifier que le réduit est toujours bien typé dans le troisième cas.

— $e = \texttt{print} \ e_1$: on a

$$\frac{\Gamma \vdash e_1 : \texttt{int} \ \& \ \phi_1}{\Gamma \vdash \texttt{print} \ e_1 : \texttt{int} \ \& \ \max(\phi_1, \texttt{P})}$$

— si c'est e_1 qui se réduit vers e'_1, alors par IR $\Gamma \vdash e'_1 : \texttt{int} \ \& \ \phi'_1$ avec $\phi'_1 \leq \phi_1$. Dès lors

$$\frac{\Gamma \vdash e'_1 : \texttt{int} \ \& \ \phi'_1}{\Gamma \vdash \texttt{print} \ e'_1 : \texttt{int} \ \& \ \max(\phi'_1, \texttt{P})}$$

avec $\max(\phi'_1, \texttt{P}) \leq \max(\phi_1, \texttt{P})$.

— sinon, c'est que e_1 est une valeur, à savoir un entier n_1, $\phi_1 = \bot$ et la réduction est $\texttt{print} \ n_1 \xrightarrow{n_1} n_1$. Le réduit n_1 est toujours bien typé de type `int` et l'effet a diminué strictement (de P à $\bot$). C'est le seul cas où l'effet diminue.

— $e = \texttt{if} \ e_1 \ \texttt{<= 0 then} \ e_2 \ \texttt{else} \ e_3$: similaire à `print`, mais sans diminution de l'effet cette fois.

— $e = \texttt{fix} \ e_1$: similaire à `print`, toujours sans diminution de l'effet et en utilisant une seconde fois la propriété de substitution.

D'une manière générale, tout se passe bien car l'effet d'une expression est monotone en les effets des sous-expressions (les deux fonctions max et $\sqcup$ sont monotones).

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 $\phi \neq \bot$.

Correction : Par récurrence structurelle sur e. S'il s'agit d'une réduction de tête, alors $e = \texttt{print} \ n$ et donc $\phi = \texttt{P}$. Sinon, il s'agit d'une réduction contextuelle $e = E(e_1) \xrightarrow{n} E(e'_1)$ avec $e_1 \xrightarrow{n} e'_1$. Par IR, on sait que l'effet ϕ_1 et e_1 n'est pas $\bot$. Pour conclure, on a besoin du résultat suivant :

Si $\Gamma \vdash e_1 : \tau_1 \ \& \ \phi_1$ et $\Gamma \vdash E(e_1) : \tau \ \& \ \phi$ alors $\phi \geq \phi_1$. On le montre par récurrence sur E.

— Si $E = \square$ c'est immédiat.

— Sinon, prenons par exemple le cas de $E = E_1 - e_2$. On a

$$\frac{\Gamma \vdash E_1(e_1) : \texttt{int} \ \& \ \phi_{11} \quad \Gamma \vdash e_2 : \texttt{int} \ \& \ \phi_2}{\Gamma \vdash E_1(e_1) - e_2 : \texttt{int} \ \& \ \phi_{11} \sqcup \phi_2}$$

Par IR sur E_1 on a $\phi_{11} \geq \phi_1$ et donc $\phi = \phi_{11} \sqcup \phi_2 \geq \phi_{11} \geq \phi_1$ par monotonie de $\sqcup$.

— Les autres cas sont similaires. Comme pour la question 8, c'est la monotonie des effets dans les règles de typage qui assure le résultat.

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 : Par applications répétées du résultat de préservation (question 8) on a $\Gamma \vdash e_i : \tau \ \& \ \phi_i$ et

$$\phi \geq \phi_1 \geq \phi_2 \cdots \geq \phi_n$$

Par la question précédente, on a $\phi_{i-1} \neq \bot$ et donc $\phi \neq \bot$.

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 $a_l \cdot a'_l = a_r \cdot a'_r$.

Correction : Par récurrence structurelle sur e.

— si la réduction de e se fait en tête, elle est forcément unique, ce qui contredit $e_l \neq e_r$;

— si les deux réductions de e se font avec le même contexte E, c'est-à-dire $e = E(\bar{e})$, $e_l = E(\bar{e}_l)$ et $e_r = E(\bar{e}_r)$ alors on peut appliquer l'IR à $\bar{e}$ et conclure par passage au contexte ;

— enfin, si les deux réductions de e ne se font pas avec le même contexte, c'est qu'il s'agit d'une soustraction ou d'une application. Supposons par exemple $e = e_1 - e_2$, avec $e_l = e'_1 - e_2$ obtenu en réduisant e_1 et $e_r = e_1 - e'_2$ obtenu en réduisant e_2. Du coup, on peut choisir $e' = e'_1 - e'_2$. Reste à montrer que les effets sont les mêmes de chaque côté. Mais on ne peut avoir simultanément $a_l \neq \emptyset$ et $a_r \neq \emptyset$, sans quoi les deux effets de e_1 et e_2 seraient différents de $\bot$ (en vertu de la question 9) et du coup l'effet de e serait $\top$ (par la définition de $\sqcup$).

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 $a = a'$.

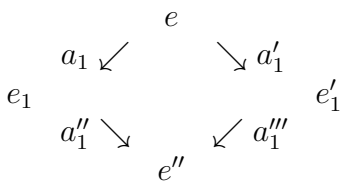
Correction : Par récurrence sur $n + n'$.

— si $n = 0$, alors e est une valeur et donc $n' = 0$ également.

— si $n > 0$, on distingue deux cas

  — si $e_1 = e'_1$, alors $a_1 = a'_1$ (admis). Par ailleurs, la préservation (question 8) garantit que e_1 a un effet $\leq \texttt{P}$. Du coup l'IR pour e_1 s'applique et permet de conclure.

— si $e_1 \neq e_1'$, alors la question précédente garantit



avec $a_1 \cdot a_1'' = a_1' \cdot a_1'''$. Par la question 4, e_1 se réduisant à la fois vers v en $n-1$ étapes et vers e'' en une étape, on en déduit que e'' se réduit vers v en $n-2$ étapes. Par IR sur e_1, la trace t de cette réduction est telle que $a_2 \cdots a_n = a_1'' \cdot t$. De même de l'autre côté, ce qui donne une trace t' de e'' vers v telle que $a_2' \cdots a_{n'}' = a_1''' \cdot t'$. Enfin, par IR sur e'', on a $t = t'$. D'où finalement

$$a_1 \cdots a_n = a_1 \cdot a_1'' \cdot t = a_1' \cdot a_1''' \cdot t' = a_1' \cdots a_{n'}'$$

---

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 :

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

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

Correction :

```
letfun f2 [fib] n =
  if n-1 <= 0 then n else fib (n-1) - (0 - fib (n-2))
letfun f1 [] fib =
  clos f2 [fib]
print ((fix (clos f1 [])) 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 :
  - 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.

Correction :

- la soustraction $c_1 - c_2$:

```
compile(E, c2)
pushq %rax
compile(E, c1)
popq  %rcx
subq  %rcx, %rax
```

- la conditionnelle `if` c_1 `<= 0 then` c_2 `else` c_3 :

```
compile(E, c1)
testq %rax, %rax
jg    L2
compile(E, c2)
jmp   L1
L2:
compile(E, c3)
L1:
```

  où L_1 et L_2 sont deux étiquettes fraîches.

- l'application c_1 c_2 : c'est le plus délicat, car il faut préserver les registres `%rdi` et `%rsi` qui sont *caller-save*

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

FIGURE 3 – Compilation vers x86-64.

```
pushq %rdi        # on sauvegarde %rdi et %rsi
pushq %rsi
compile(E, c1)    # on évalue la fonction
pushq %rax        # et on la pose sur la pile
compile(E, c2)    # on évalue l'argument
movq  %rax, %rdi
popq  %rsi        # on récupère la fonction
call  *(%rsi)     # on fait l'appel
popq  %rsi        # on restaure %rsi et %rdi
popq  %rdi
```

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.

Correction : Il s'agit là de registres *caller-save*, qui peuvent être écrasés par l'appel à `malloc`.

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

Correction :

```
_fix:
        pushq %rdi           # sauvegarde l'argument de l'appel
        movq  %rsi, %rdi     # la fermeture fix devient l'argument
        movq  8(%rsi), %rsi  # et sa valeur contenue devient la fermeture
        call  *(%rsi)        # on appelle
        movq  %rax, %rsi     # et le résultat est une fermeture
        popq  %rdi           # on récupère l'argument initial
        jmp  *(%rsi)         # et on fait l'appel (ici terminal)
```

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

Correction : Cela permet de compiler simplement l'application, sans faire de cas particulier pour un point fixe. Par ailleurs, il faudrait faire cette distinction à l'exécution, car on ne peut savoir, de manière générale, si la valeur appliquée a été obtenue avec `fun` ou `fix`.

```
(if ... then (fun x -> ...) else (fix ...)) 42
```

---

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.

Correction : Cette grammaire est ambiguë, car il y a deux arbres de dérivation possibles pour

```
fun x -> 1 - 2
```

à savoir `(fun x -> 1) - 2` d'une part et `fun x -> (1 - 2)` d'autre part. Il y a d'autres ambiguïtés, comme `1-2-3`. Elle n'est donc pas LR(1).

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

Correction : Les ambiguïtés proviennent des règles se terminant par E, c'est-à-dire `let`, `if`, `fun` et la soustraction, avec à la suite, soit $-E$, soit S. Il y a plusieurs solutions. Le plus naturel est de décider

— que les constructions `let`, `if`, `fun` sont moins fortes que la soustraction (ainsi la soustraction reste dans le corps du `let`, de la fonction ou dans la branche `else`, ce qui est naturel) ;
— que la soustraction est associative à gauche ;
— que l'application est plus forte que `let`, `if`, `fun` (comme en OCaml, où l'application a la priorité la plus forte), en favorisant la lecture de `(`, `ident` et `constante` ;

c'est-à-dire au final

```
%nonassoc IN ELSE ARROW
%left     MINUS
%nonassoc LPAR IDENT CONSTANT (* c'est-à-dire first(S) *)
```

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`.