Fondements de la programmation

Exercices 11 : extensions, type safety, machine de Krivine

On se donne un ensemble L de mots ou étiquettes. Un **type somme** est la donnée d'un dictionnaire fini dont les clés sont des mots et dont les valeurs sont des types : $\{\text{mot}_1 : T_1, \ldots, \text{mot}_n : T_n\}$. On étend les types avec les types sommes et les termes avec le pattern-matching.

$$
\begin{aligned}
T &:= \ldots \mid \{\text{mot}_1 : T_1, \ldots, \text{mot}_n : T_n\} \\
t &:= \ldots \mid \texttt{match } t \texttt{ with} \\
&\qquad \texttt{case } \text{mot}_1\ x_1 \Rightarrow t_1 \\
&\qquad \vdots \\
&\qquad \texttt{case } \text{mot}_n\ x_n \Rightarrow t_n \\
&\quad \mid \text{mot } t \texttt{ as } T \\
v &:= \ldots \mid \text{mot } t \texttt{ as } T
\end{aligned}
$$

Remarque : sans ce « as T » on perdrait l'unicité du type associé à un terme (on aurait par exemple Vrai unit de type $\{\text{Vrai} : \texttt{Unit}\}$ mais aussi de type $\{\text{Vrai} : \texttt{Unit}, \text{Faux} : \texttt{Unit}\}$, ou de type $\{\text{Vrai} : \texttt{Unit}, \text{Saispas} : \texttt{Unit}\}$ etc.).

Règles de réduction.

$$\frac{t \to t'}{\texttt{match } t \texttt{ with} \ldots \to \texttt{match } t' \texttt{ with} \ldots}$$

$$\frac{t \to t'}{\text{mot } t \texttt{ as } T \to \text{mot } t' \texttt{ as } T}$$

$$
\frac{}{\begin{array}{l}\texttt{match } \text{m}_i\ t \texttt{ as } T \texttt{ with} \\ \qquad \texttt{case } \text{mot}_1\ x_1 \Rightarrow t_1 \\ \qquad \vdots \\ \qquad \texttt{case } \text{mot}_n\ x_n \Rightarrow t_n \\ \to t_i[x_i := t]\end{array}}
$$

Règles de typage.

$$\frac{\Gamma \vdash t : \{\ldots, m_i : T_i, \ldots\} \qquad \Gamma, x_i : T_i \vdash t_i : T \quad (\forall i)}{\begin{array}{c}\Gamma \vdash \texttt{match } t \texttt{ as } \{\ldots, m_i : T_i, \ldots\} \texttt{ with} \\ \ldots \texttt{ case } m_i\ x_i \Rightarrow t_i \ldots \quad : T\end{array}}$$

$$\frac{\Gamma \vdash t_i : T_i}{\Gamma \vdash \text{m}_i\ t_i \texttt{ as } \{\ldots, m_i : T_i, \ldots\} : \{\ldots, m_i : T_i, \ldots\}}$$

Théorème de préservation. Si $\Gamma \vdash t : T$ et $t \to t'$, alors $\Gamma \vdash t' : T$. Ceci est valable pour le lambda-calcul simplement typé comme pour les extensions présentées ici. La réciproque est en générale fausse.

Théorème de progression. Si $\vdash t : T$ (notez le contexte vide), alors soit t est une valeur, soit il existe t' tel que $t \to t'$.

Type safety. Un langage est dit sûr au niveau du typage (*type safe*) lorsqu'il possède les deux propriétés précédentes (préservation et progression), éventuellement en élargissant la propriété de progression de façon à permettre au calcul de terminer sur des erreurs (des exceptions) aussi bien que sur des valeurs. Notez que la progression n'implique pas la terminaison.

Théorème de normalisation forte. En lambda-calcul simplement typé la β-réduction termine toujours (sans avoir besoin d'utiliser une stratégie particulière).

La **machine abstraite de Krivine (KAM)** implémente l'appel par nom en λ-calcul. On définit 4 notions :

terme $t := x \mid (t\ t) \mid \lambda x.t$

environnement $e := \emptyset \mid e, x : c$ (un dictionnaire associant des clôtures à des variables)

clôture $c := (t, e)$

pile $\pi := \varepsilon \mid c :: \pi$

L'état de la machine est représenté par un triplet fait d'un terme d'un environnement et d'une pile. On se donne un terme initial (un programme) avec un environnement et une pile vide. Une étape de réduction fait passer d'un état à l'état suivant en appliquant trois règles :

$$\frac{(t\ u) \quad e \quad \pi}{t \quad e \quad (u, e) :: \pi}\ \text{push}$$

$$\frac{\lambda x.t \quad e \quad c : \pi}{t \quad e, x :: c \quad \pi}\ \text{pop}$$

$$\frac{x \quad e, x : (t, e') \quad \pi}{t \quad e' \quad \pi}\ \text{deref}$$

Par exemple, l'évaluation du terme $((\lambda x.x)\ y)$ sur la KAM par nom se déroule ainsi :

$$\cfrac{\cfrac{\cfrac{((\lambda x.x)\ y) \quad \emptyset \quad \varepsilon}{\lambda x.x \quad \emptyset \quad (y, \emptyset)}\ \text{push}}{x \quad x : (y, \emptyset) \quad \varepsilon}\ \text{pop}}{y \quad \emptyset \quad \varepsilon}\ \text{deref}$$

Le résultat est y (plus aucune règle ne s'applique).

La **KAM par valeur** est définie en modifiant les piles de façon à ce qu'elles contiennent deux sortes d'éléments (*fonctions* et *arguments*) :

pile $\pi = \varepsilon \mid Fc :: \pi \mid Ac :: \pi$

Les règles sont, par priorités décroissantes :

$$\frac{(t\ u) \quad e \quad \pi}{t \quad e \quad A(u,e) :: \pi}\ \text{push}$$

$$\frac{\lambda x.t \quad e \quad A(u,e') :: \pi}{u \quad e' \quad F(\lambda x.t, e) :: \pi}\ \text{swap}$$

$$\frac{x \quad e, x = (t, e') \quad \pi}{t \quad e' \quad \pi}\ \text{deref}$$

$$\frac{v \quad e \quad F(\lambda x.t, e')\pi}{t \quad e', x =: (v, e) \quad \pi}\ \text{pop}$$

Dans cette dernière règle, v désigne une valeur c'est à dire ici un lambda-terme qui n'est pas une application (donc une variable ou un lambda).

Par exemple, l'évaluation du terme $((\lambda x.x)\ y)$ sur la KAM par valeur se déroule ainsi :

$$\frac{\dfrac{\dfrac{((\lambda x.x)\ y) \quad \emptyset \quad \varepsilon}{\lambda x.x \quad \emptyset \quad A(y,\emptyset)}\ \text{push}}{y \quad \emptyset \quad F(\lambda x.x, \emptyset)}\ \text{swap}}{x \quad x = (y, \emptyset) \quad \varepsilon}\ \text{pop}$$
$$y \quad \emptyset \quad \varepsilon\ \text{deref}$$

1 Exercices

Question A. Préservation. Prouver le théorème de préservation pour le λ-calcul simplement typé.

Question B. Normalisation forte. Pourquoi ne prouve t'on pas le théorème de normalisation forte du lambda-calcul simplement typé par induction sur les termes ?

Question C. Morris. En C que vaut Morris(1, 0) ? Citer un langage dans lequel cette fonction s'évaluerait différemment.

```
int Morris(int a, int b)
{
  if (a == 0) {
    return 1;
  }
  else {
    return Morris(a - 1, Morris(a, b));
  }
}
```

Question D. Scala. En Scala, `val` est l'équivalent du `let` de Caml, il permet d'introduire une variable immutable dont la valeur est obtenue en appel par valeur, `var` permet d'introduire une variable mutable dont la valeur est également fixée en appel par valeur, `def` permet d'introduire une variable immutable en appel par nom et `lazy val` est une variante de `val` qui est évaluée paresseusement, en appel par nécessité. Un bloc d'instructions (entre accolades) s'évalue comme la dernière valeur du bloc. Sauriez vous dire quelle ligne de code provoque quel affichage dans l'extrait de programme Scala suivant ?

```
val x = {println("x"); 1}
var y = {println("y"); 2}
def z = {println("z"); 3}
lazy val w = {println("w"); 4}
x + x
y + y
z + z
w + w
```

Question E. KAM par nom. Donner l'exécution de la KAM par nom sur le terme $(((\lambda xy.x)\ z)\ z')$.

Question F. KAM par valeur. Donner l'exécution de la KAM par valeur sur le terme $(((\lambda xy.x)\ z)\ z')$.

Question G. KAM (partiel 2015). On veut comparer le temps d'exécution de la KAM par nom et de la KAM par valeur. Pour mesurer ce temps on compte le nombre de règles appliquées dans chaque exécution mais **sans compter** les applications la règle *swap*. Ainsi les exécutions des deux machines sur le terme $((\lambda x.x)\ y)$ prennent autant de temps (3 règles hors règle *swap*).

1. Est-ce encore le cas sur le terme $(((\lambda xy.x)\ z)\ z')$?
2. Les deux machines ont-elles toujours les mêmes temps d'exécution ou bien pouvez-vous trouver un terme pour lequel la KAM par nom est plus rapide et/ou un terme pour lequel la KAM par valeur est plus rapide ? Justifier par un raisonnement ou en donnant des exemples de termes et leurs exécutions.