Évaluation des programmes mini-ML

L'évaluation d'une expression (donc d'un programme) est une fonction partielle des expressions vers les *valeurs*.

Le fait que la fonction soit partielle modélise le fait qu'une évaluation peut ne pas terminer ou s'interrompre pour cause d'erreur.

Les valeurs v de mini-ML sont presqu'un sous-ensemble strict des expressions défini récursivement par les cas suivants :

- unité ou constante entière — $()$ ou 0 ou 1 ou 2 etc.
- fermeture — $\langle \textbf{fun}\ x \to e, \rho \rangle$
  où ρ est un environnement.
  Pour les opérateurs : $\langle (+), \rho \rangle$ etc.

Fermeture, liaison et environnement (le retour)

Une *fermeture* est une sorte de paire faite d'une fonction et d'un environnement. Cela signifie en particulier que les fonctions font partie des valeurs, c.-à-d. qu'elles peuvent être le résultat de l'évaluation d'un programme mini-ML (cf. exemple page 12) : *c'est la caractéristique d'un langage fonctionnel*.

Pour définir l'évaluation nous devons modifier le concept de liaison : une liaison associe maintenant une variable à une *valeur* (et non plus une expression).

L'évaluation (donc peut-être la valeur) dépendra de l'environnement au lieu où se trouve l'expression.
Notons $(x \mapsto v) \oplus \rho$ l'ajout d'une liaison $x \mapsto v$ dans ρ (en tête).

Évaluation des programmes mini-ML (suite)

Pour chaque expression e nous définissons une règle d'évaluation dans un environnement ρ qui donne la valeur v :

- x — Chercher la *première* valeur associée à x dans ρ.
- **fun** $x \to e$ — La valeur est $\langle \textbf{fun } x \to e, \rho \rangle$.
- `+` `-` `/` `*` — La valeur est $\langle (+), \rho \rangle$ etc.
- e_1 + e_2 etc. — Évaluer e_1 et e_2 dans ρ et sommer, etc.
- () ou 0 ou 1 ou 2 etc. — La valeur est () ou 0 ou 1 etc.
- (e) — Évaluer e dans ρ.

Évaluation des programmes mini-ML (suite)

- **let** $x = e_1$ **in** e_2 — Évaluer e_1 en v_1 dans ρ ;
  évaluer e_2 en v dans $(x \mapsto v_1) \oplus \rho$.
- $e_1\ e_2$ — Évaluer e_1 et e_2 en v_1 et v_2 dans ρ
  (l'ordre n'est pas spécifié) ;
  v_1 doit être de la forme $\langle \textbf{fun}\ x \to e, \rho' \rangle$;
  évaluer e dans $(x \mapsto v_2) \oplus \rho'$: la valeur est v.

Exemple d'évaluation

L'environnement est initialement vide : $\rho = \{\}$. Évaluons

```
let x = 0 in
  let id = fun x -> x in
  let y = id (x) in
  let x = (fun x -> fun y -> x + y) 1 2
in x+1;;
```

- L'évaluation de `let x = 0 in ...` impose d'évaluer d'abord `0` qui vaut 0. Ensuite on crée la liaison $\mathsf{x} \mapsto 0$, on l'ajoute à ρ, ce qui donne $\{\mathsf{x} \mapsto 0\}$, et on évalue la sous-expression élidée dans ce nouvel environnement.
- L'évaluation de `let id = fun x -> x in ...` se fait dans l'environnement $\{\mathsf{x} \mapsto 0\}$. La valeur v est alors la fermeture $\langle \textbf{fun } \mathsf{x} \to \mathsf{x}, \{\mathsf{x} \mapsto 0\}\rangle$. On étend l'environnement courant avec $\mathsf{id} \mapsto v$ et on évalue la sous-expression avec.

Exemple d'évaluation (suite)

- L'évaluation de `let y = id (x) in ...` se fait dans l'environnement $\{\mathsf{id} \mapsto \langle \textbf{fun } \mathsf{x} \to \mathsf{x}, \{\mathsf{x} \mapsto 0\}\rangle; \mathsf{x} \mapsto 0\}$.
  - On évalue d'abord `id(x)` dans l'environnement courant. Pour cela on évalue `id` et `x` séparément. Ce sont tous deux des variables, donc on cherche dans l'environnement la première liaison de même nom. La valeur de `id` est donc $\mathsf{id} \mapsto \langle \textbf{fun } \mathsf{x} \to \mathsf{x}, \{\mathsf{x} \mapsto 0\}\rangle$ et celle de `x` est 0. Il faut évaluer `x` dans l'environnement $(\mathsf{x} \mapsto 0) \oplus \{\mathsf{x} \mapsto 0\}$, ce qui donne 0.
  - On crée la liaison $\mathsf{y} \mapsto 0$, on l'ajoute à l'environnement courant et on évalue la sous-expression élidée avec le nouvel environnement.
- L'évaluation de `let x = (fun x -> fun y -> x + y) 1 2 in ...` se fait dans l'environnement
  $\{\mathsf{y} \mapsto 0; \mathsf{id} \mapsto \langle \textbf{fun } \mathsf{x} \to \mathsf{x}, \{\mathsf{x} \mapsto 0\}\rangle; \mathsf{x} \mapsto 0\}$.

Évaluation formelle des programmes mini-ML

Si on note $[\![e]\!]\rho$ la valeur obtenue en évaluant l'expression e dans l'environnement ρ, alors nous pouvons résumer l'évaluation des programmes mini-ML ainsi :

$$
\begin{aligned}
[\![\overline{n}]\!]\rho &= \dot{n} \quad \text{où } \overline{n} \text{ est un entier mini-ML et } \dot{n} \in \mathbb{N} \\
[\![e_1 + e_2]\!]\rho &= [\![e_1]\!]\rho + [\![e_2]\!]\rho \quad \text{etc.} \\
[\![(e)]\!]\rho &= [\![e]\!]\rho \\
[\![x]\!]\rho &= \rho(x) \quad \text{(la première liaison de } x \text{ dans } \rho) \\
[\![\textbf{fun } x \to e]\!]\rho &= \langle \textbf{fun } x \to e, \rho \rangle \\
[\![\textbf{let } x = e_1 \textbf{ in } e_2]\!]\rho &= [\![e_2]\!]((x \mapsto [\![e_1]\!]\rho) \oplus \rho) \\
[\![e_1\ e_2]\!]\rho &= [\![e]\!]((x \mapsto [\![e_2]\!]\rho) \oplus \rho') \\
&\qquad \text{où } [\![e_1]\!]\rho = \langle \textbf{fun } x \to e, \rho' \rangle
\end{aligned}
$$

L'évaluation consiste à appliquer les équations de la gauche vers la droite jusqu'à la terminaison (s'il y a) ou une erreur (à l'exécution).

Exemples

On établit directement que $[\![(\textbf{fun}\ x \to e_2)\ e_1]\!]\rho = [\![\textbf{let}\ x = e_1\ \textbf{in}\ e_2]\!]\rho$, c.-à-d. que la liaison locale n'est pas nécessaire en théorie.

Reprenez la description informelle de l'évaluation du programme page 27 et donnez-en une description formelle.

Pour mieux comprendre la nécessité des fermetures, décrivez l'évaluation de

```
let x = 1 in
  let f = fun y -> x + y in
  let x = 2
in f(x)
```

et du programme page 11.

Exercices

- Quelle différences y a-t-il entre
  - $f\ x\ y$
  - $(f\ x)\ y$
  - $f\ (x\ y)$
  - $(f\ (x))\ (y)$
- Quelle différences y a-t-il entre
  - f
  - **fun** $x \to f\ x$
  - **fun** $x\ y \to f\ x\ y$

Solution

La seconde expression retarde l'évaluation jusqu'à ce que le premier argument soit fourni.

La dernière retarde l'évaluation jusqu'à ce que les deux arguments soient fournis.

Cela peut délaisser ou dupliquer certains calculs :

```
let f x = let z = print_int x in fun y -> x + y
let test g = let h = g(1) in h(2) + h(3)
# test f;;
1- : int = 7
# test (fun x y -> f x y);;
11- : int = 7
```

Applications partielles et complètes

Si les fonctions sont des valeurs, elles peuvent être retournées en résultat (c.-à-d. être la valeur d'une application). Par exemple :

```
let add = fun x -> fun y -> x + y in
  let incr = add 1                          (* incr est une fonction *)
in incr 5;;
```

On parle d'application *partielle*, par opposition à application *complète*, qui ne retourne pas de fonction, comme (`add 1 5`). Les opérateurs peuvent aussi être utilisés en position préfixe dans une expression en les parenthèsant : (+) 1 2. Comme toute application, les opérations peuvent être aussi partiellement évaluées :

```
let incr = (+) 1  (* incr est une fonction *)
in incr 5;;
```

Non-terminaison

En théorie, nous pouvons d'ores et déjà calculer avec mini-ML tout ce qui est calculable avec l'ordinateur sous-jacent. Par exemple, nous avons déjà la récurrence, comme le montre le programme suivant qui ne termine pas :

```
let omega = fun f -> f f in omega omega
```

Cela se manifeste par

$$
\begin{aligned}
&[\![\texttt{let omega = fun f -> f f in omega omega}]\!]\rho \\
&\quad = [\![\texttt{omega omega}]\!]((\texttt{omega} \mapsto [\![\texttt{fun f -> f f}]\!]\rho) \oplus \rho) \\
&\quad = [\![\texttt{f f}]\!]((\texttt{f} \mapsto \langle \texttt{fun f -> f f}, \rho \rangle) \oplus \rho) \\
&\quad = \textit{idem}
\end{aligned}
$$

Fonctions récursives

Pour mettre en évidence la puissance de mini-ML, voyons comment définir des fonctions récursives à l'aide de la fonction auto-applicative `omega`.

Définissons d'abord une fonction `fix`, traditionnellement appelée le *combinateur Y de point fixe de Curry* :

```
let omega = fun f -> f f;;
let fix = fun g -> omega (fun h -> fun x -> g (h h) x);;
```

Point fixe d'une fonction

On démontre (péniblement) que pour toute fonction f et variable x :

$$[\![(\text{fix } f)\, x]\!]\rho = [\![f\ (\text{fix } f)\, x]\!]\rho$$

En d'autre termes, pour tout x on a $(\text{fix } f)\ x \equiv f\ (\text{fix } f)\ x$, soit $(\text{fix } f) \equiv f\ (\text{fix } f)$

D'autre part, par définition, le point fixe p d'une fonction f vérifie $p = f(p)$. Donc le point fixe d'une fonction f, *s'il existe*, est (fix f).

Il est possible de définir la sémantique (c.-à-d. l'évaluation) d'une famille d'opérateurs de point fixes (et non d'un seul comme précédemment) en posant qu'un tel opérateur doit satisfaire

$$[\![\text{fix } e]\!]\rho = [\![e_1]\!](f \mapsto [\![\text{fix}\,(\textbf{fun}\, f \to e_1)]\!]\rho \oplus \rho')$$

où $[\![e]\!]\rho = \langle \textbf{fun}\, f \to e_1, \rho' \rangle$

Factorielle et cas général

```
let pre_fact =
  fun f -> fun n -> if n=1 then 1 else n * f(n-1);;
let fact = fix pre_fact;;
```

Donc `fact` est le point fixe de `pre_fact`, s'il existe, c'est-à-dire

$$\begin{aligned}[\![\texttt{fact}]\!]\rho &= [\![\texttt{pre\_fact (fact)}]\!]\rho \\ &= [\![\texttt{fun n -> if n=1 then 1 else n * fact(n-1)}]\!]\rho\end{aligned}$$

Donc `fact` est la fonction factorielle (équation de récurrence). On peut alors prédéfinir un opérateur de point fixe `fix` (qui n'est pas forcément celui de Curry) et permettre au programmeur de s'en servir directement, mais nous allons plutôt doter mini-ML d'une liaison récursive native :

$$[\![\textbf{let rec } f = e_1 \textbf{ in } e_2]\!]\rho = [\![\textbf{let } f = \mathsf{fix}\,(\textbf{fun } f \to e_1) \textbf{ in } e_2]\!]\rho$$