Compilation – TP4 : *mini-Ocaml*

Année universitaire 2013-2014

1 Introduction

Nous allons réaliser un interpréteur d'un sous-ensemble de *Ocaml* en utilisant la notion d'**environnement**. D'autres techniques sont possibles, mais celle-ci reste simple et raisonnablement efficace. Nous allons écrire un analyseur lexical et un analyseur syntaxique pour notre *mini-Ocaml* permettant d'obtenir l'arbre syntaxique abstrait d'une expression *Ocaml*. Nous écrirons aussi une fonction d'évaluation prenant en argument un environnement et l'arbre syntaxique abstrait d'une expression *Ocaml*, et fournissant en résultat la valeur de cette expression.

2 Évaluation à l'aide d'environnement

Un environnement peut être vu comme une table dans laquelle certaines variables sont liées à des valeurs. Lorsque l'on doit évaluer une expression de la forme `((function x → e) v)`, plutôt que de substituer `v` à `x` dans `e`, on évalue `e` dans un environnement où `x` est lié à `v`.
Dans ce modèle de calcul avec environnements, les valeurs fonctionnelles posent un problème particulier car elles peuvent contenir des variables libres qui n'ont de sens que par rapport à un environnement. Par exemple :

```
let x = 1 in function y -> x + y
```

on voit que la fonction `function y → x + y` dépend de son environnement de définition pour donner une valeur à `x`. La valeur d'une expression fonctionnelle dans un environnement sera donc une paire formée de l'expression et de cet environnement. Une telle paire s'appelle une **fermeture** (ou **closure** en anglais). Pour l'exemple précédent, l'évaluation de l'expression : `let x = 1 in function y → x + y` donnera la paire `([(x, 1)], function y → x + y)`. Le premier élément de la paire est une liste d'association qui représentera l'environnement. Prenons un autre exemple, considérons l'expression : `(function x → function y → x*y) 3`. Dans un calcul par substitution, sa valeur est : `(function y → 3*y)`. Dans un calcul avec environnement, sa valeur sera la paire : `([(x, 3)], function y → x*y)`.

L'évaluation d'une expression dans un environnement sera définie par une relation ternaire notée :

$$E \vdash e \Rightarrow v$$

et signifiant « l'expression e à la valeur v dans l'environnement E ».

Pour chaque construction du langage, on donne une règle de déduction. Les règles suivantes correspondent à l'évaluation dite par valeur.

$$\frac{E(x) = v}{E \vdash x \Rightarrow v}(\text{Var})$$

$$\frac{E \vdash e_1 \Rightarrow v_1 \qquad \ldots \qquad E \vdash e_n \Rightarrow v_n}{E \vdash (e_1, \ldots, e_n) \Rightarrow (v_1, \ldots, v_n)}(\text{tuple})$$

$$\frac{E \vdash e_1 \Rightarrow v_1 \qquad \ldots \qquad E \vdash e_n \Rightarrow v_n}{E \vdash e_1 :: \ldots :: e_n :: [] \Rightarrow v_1 :: \ldots :: v_n :: []}(\text{list})$$

$$\frac{E \vdash e_1 \Rightarrow true \qquad E \vdash e_2 \Rightarrow v}{E \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Rightarrow v}(\text{Cond}_1)$$

$$\frac{E \vdash e_1 \Rightarrow false \qquad E \vdash e_3 \Rightarrow v}{E \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Rightarrow v}(\text{Cond}_2)$$

$$\frac{}{E \vdash (\text{function } p_1 \ \rightarrow e_1 | \ \ldots | \ p_n \rightarrow e_n) \Rightarrow (E, \text{function } p_1 \ \rightarrow e_1 | \ \ldots | \ p_n \rightarrow e_n)}(\text{Fun})$$

où p_i est un motif du type `ml_pattern` défini dans le fichier `ast.ml` :

```
type ml_pattern =
| Ml_pattern_var of string
| Ml_pattern_bool of bool
| Ml_pattern_int of int
| Ml_pattern_pair of ml_pattern * ml_pattern
| Ml_pattern_nil
| Ml_pattern_cons of ml_pattern * ml_pattern
```

La règle définissant l'application d'une fonction à un argument est donnée ci-dessous :

$$\frac{E \vdash \text{expr}_1 \Rightarrow (E', \text{function } p_1 \rightarrow e_1 | \ \ldots | \ p_n \rightarrow e_n) \quad E \vdash \text{expr}_2 \Rightarrow v_2 \quad (\text{unify}(p_j, v_2)@E') \vdash e_j \Rightarrow v}{E \vdash (\text{expr}_1 \ \text{expr}_2) \Rightarrow v}(\text{App})$$

dans cette règle, p_j est le premier pattern qui s'unifie avec v_2 (aucun p_i avec $i < j$ ne s'unifie avec v_2). On voit ici que lorsque l'argument « match » un motif d'une fonction, de nouvelles liaisons sont créées dans l'environnement. Par exemple l'application suivante :

```
(function [] -> 0 | hd :: tl -> hd + 1) (2 :: 3 :: 4 :: [])
```

va ajouter les liaisons : `(hd, 2)` et `(tl, 3 :: 4 :: [])` à l'environnement de la fermeture de `(function [] → 0 | hd :: tl → hd + 1)` (qui est ici `([], (function [] → 0 | hd :: tl → hd + 1))`) pour évaluer le corps de l'alternative `hd + 1`.

$$\frac{E \vdash e_1 \Rightarrow v_1 \qquad (x, v_1) :: E \vdash e_2 \Rightarrow v}{E \vdash \text{let } x = e_1 \text{ in } e_2 \Rightarrow v}(\text{Let})$$

Pour la prise en compte de la récursivité, nous allons utiliser la possibilité de définir des structures de données circulaires en *Ocaml*. Par exemple, on peut définir la liste infinie : `let rec l = 1 :: l` en *Ocaml*. Étant donné un environnement `E`, un nom de fonction `f` et une expression fonctionnelle `function` $p_1 \rightarrow e_1$ `|` `...` `|` $p_n \rightarrow e_n$ (où les e_i peuvent contenir des occurences de `f`), nous construirons un environnement `E'` tel que `E' = (f, (E', function` $p_1 \rightarrow e_1$ `|` `...` `|` $p_n \rightarrow e_n$`))` :: `E`. La règle permettant de prendre en compte la récursivité est alors la suivante :

$$\frac{E' = (f, (E', function\ p_1 \rightarrow e_1 \mid \ldots \mid p_n \rightarrow e_n)) :: E \vdash e_2 \Rightarrow v}{E \vdash \text{let } f = function\ p_1 \rightarrow e_1 \mid \ldots \mid p_n \rightarrow e_n \text{ in } e_2 \Rightarrow v}(\text{Letrec})$$

3 Écriture d'un évaluateur en *OCaml*

Une description de l'évaluation de *mini-Ocaml* par une fonction d'évaluation écrite en *Ocaml* est d'une certaine façon une tricherie, car elle suppose que l'on sait déjà évaluer la fonction d'évaluation. Cependant, elle est très instructive car elle permet de ramener l'évaluation de toutes les expressions du langage à l'évaluation d'un petit nombre de fonctions simples. Pour décrire de façon complète l'implémentation du langage, il suffit ensuite de montrer comment la fonction d'évaluation peut-être elle même évaluée ou compilée en langage machine.

Nous commençons par introduire un type `ml_expr` décrivant l'arbre syntaxique abstrait de notre *mini-Ocaml* et un type `ml_val` décrivant les valeurs qui pourront être résultat d'un calcul. Dans les fermetures, les environnements sont représentés par des listes d'association de type `(string * ml_val) list`. Ces types sont définis dans les fichiers `ast.ml` et `value.ml`.

Les fonctions d'accès aux paires (`Ml_fst` et `Ml_snd`) sont prédéfinies (fichier `ast.ml`) de même que les opérations binaires arithmétiques (`Ml_add` $(+)$, `Ml_sub` $(-)$, `Ml_mult` $(\times)$) et les opérations de comparaison (`Ml_eq` $(=)$ et `Ml_less` $(<)$) définis dans le fichier `ast.ml`.

4 Travail à réaliser

Un squelette vous est fourni (`miniocaml.tgz` sous `commun`). Vous pouvez d'ors et déjà compiler et vous obtiendrez un exécutable `main.native`. Vous pouvez le lancer avec l'option `-d` qui permet de réécrire l'arbre syntaxique abstrait de l'expression que vous avez tapée (ceci permet de tester votre analyseur syntaxique).

Vous devez compléter les fichiers : `lexer.mll`, `parser.mly` et `eval.ml`.