Programmation TP 1
Mini-ML : évaluation d'expressions

{ jrobert, fbouchez, apardon } @ens-lyon.fr
http://perso.ens-lyon.fr/florent.bouchez/

18 Septembre 2007

Dans ce TP, nous allons définir un évaluateur pour un tout petit langage fonctionnel : "mini-ML". On commence par définir un évaluateur d'expressions arithmétiques, que l'on étend ensuite par une construction "let-in". Finalement, on ajoute les fonctions à notre langage.

Emacs, Tuareg

On va utiliser l'interprète `ocaml`, dans l'éditeur de texte `emacs`, qui admet un mode nommé `tuareg` facilitant le travail sous *OCaml*. Voici les raccourcis essentiels[1] :

`C-x C-f`	Ouvrir un nouveau fichier.
`C-x C-s`	Sauvegarder le fichier courant.
`C-x b`	Changer de fichier courant.
`C-h C-h`	Aide sur l'aide d'emacs.
`C-c C-e`	Évaluer la phrase courante dans ocaml.
`C-c C-r`	Évaluer la région courante dans ocaml.
`C-c C-k`	Interrompre ocaml (quand ça boucle).
`C-c h`	Afficher l'aide *Ocaml* sur un symbole de la librairie standard.
`C-c C-h`	Afficher les raccourcis de `tuareg`.

1 Expressions Arithmétiques

Le langage de base est très simple, on le décrit par le type de données suivant :

```
type expr =
  | TInt of int                    (* i *)
  | TAdd of expr * expr            (* e1+e2 *)
  | TMul of expr * expr            (* e1*e2 *)
  | TDiv of expr * expr            (* e1/e2 *)
```

Q 1.1 Écrivez une fonction d'évaluation pour ces termes : `eval: expr →int`.

[1] si vous tenez à bosser sous `vi`, débrouillez-vous !

Q 1.2 On voudrait compter le nombre de divisions effectuées lors de l'évaluation d'une expression. Définissez une seconde fonction d'évaluation : `eval_count: expr →int`, qui utilise une référence pour compter le nombre de divisions, et qui affiche ce nombre à l'aide des fonctions `print_int`, `print_string` et `print_newline`.

Q 1.3 Définissez une exception `Div_by_Zero`, que vous lèverez en cas de division par zéro.

2 "Let in", Environnements

On veut rajouter une construction "let-in" à notre langage, c'est à dire la possibilité de retenir une valeur dans un environnement. Dans l'exemple suivant, l'expression `x*x-y*y` va être évaluée dans un environnement indiquant les valeurs des variables x et y.

```
let x=2+3 in
let y=5*3 in
  x*x-y*y
```

On va représenter les variables par des chaînes de caractères (`string`). On peut alors définir un environnement par le type et les valeurs suivantes, où `'a` représente le type des valeurs contenues dans l'environnement :

```
type 'a env
val empty_env: 'a env
val get_env: 'a env → string → 'a
val add_env: 'a env → string → 'a → 'a env
```

Remarquez qu'il s'agit là d'environnements *fonctionnels* : la fonction `add_env` ne modifie pas l'environnement qu'on lui passe, mais en renvoie un nouveau.

Q 2.1 La solution la plus naturelle consiste à utiliser une liste de couples pour représenter un environnement : **`type`** `'a env = (string * 'a) list`. Écrivez les fonctions `get_env` et `add_env` pour ce type d'environnements.

Q 2.2 Mais on peut aussi utiliser les fonctions d'OCaml : **`type`** `'a env = string →'a`. Même question avec ce type là.

Q 2.3 Dans les deux questions précédentes, levez une exception `Unbound_Variable` lorsque la variable recherchée n'apparaît pas dans l'environnement.

On peut maintenant ajouter à notre langage la construction "let-in", ainsi que des variables :

```
type expr =
  | TInt of int                      (* i *)
  | TAdd of expr * expr              (* e1+e2 *)
  | TMul of expr * expr              (* e1*e2 *)
  | TDiv of expr * expr              (* e1/e2 *)
  | TLet of string * expr * expr     (* let x = e1 in e2 *)
  | TVar of string                   (* x *)
```

Q 2.4 Écrivez la nouvelle fonction d'évaluation : `eval: int env →expr →int`.

Q 2.5 Vérifiez son comportement sur des exemples. Que se passe-t'il en particulier pour l'expression suivante : **`let`** `x = 5` **`in`** `(`**`let`**` x = 3 `**`in`**` x) * x` ?

3 Fonctions et Application

Et maintenant, on rajoute deux constructions : les fonctions, et l'application :

```
type expr =
  | TInt of int                    (* i *)
  | TAdd of expr * expr            (* e1+e2 *)
  | TMul of expr * expr            (* e1*e2 *)
  | TDiv of expr * expr            (* e1/e2 *)
  | TLet of string * expr * expr   (* let x = e1 in e2 *)
  | TVar of string                 (* x *)
  | TFun of string * expr          (* fun x → e *)
  | TApp of expr * expr            (* e1(e2) *)
```

Désormais, la fonction d'évaluation ne renvoie pas forcément un entier : elle peut aussi renvoyer une valeur *fonctionnelle* (considérer l'évaluation de `TFun("x",TMul(TVar "x",TVar "x"))`.

Q 3.1 Proposez un type de retour pour la fonction d'évaluation (`type value = ...`).

Q 3.2 Écrivez la nouvelle fonction d'évaluation : `eval: value env →expr →value`. Éventuellement, revenez sur votre définition des valeurs...

Q 3.3 Vérifiez que votre fonction évalue correctement le terme suivant :
`(let y=3 in fun x →x*y) 5`.

Q 3.4 *(Subsidiaire)* et le "let rec" ?