Fouine

Victor Boone - Gabrielle Pauvert

Mars 2018

1 Généralités

Dans notre implémentation de *Fouine*, il y a 4 grands types

Composante	Description	Type
Expression	Ce qu'on évalue	`expr_f`
Valeurs	Ensemble des valeurs pouvant être renvoyées	`val_f`
Environnement	Stocke des couples (variables, valeurs)	`env_f`
Mémoire	Représentation de la mémoire en Fouine	`mem_f`

On ne parlera pas de *programme* en Fouine, mais plutôt d'*expression*. Tout est expression qu'on cherche à évaluer. L'évaluation des fonctions est faite par la fonction `eval` :

```
val eval : expr_f -> env_f -> mem_f -> val_f * mem_f
```

La mémoire associée à un programme fouine n'est pas vue comme un contexte qui pourra être modifié par effet de bord. Elle est passée en argument à la fonction d'évaluation, qui renvoie la valeur de l'expression évaluée **et** le nouvel état de la mémoire. Il en va différemment de l'environnement ; celui-ci est passé en paramètre et agit comme un "contexte" dans lequel s'évalue l'expression, mais ne peut être modifié de manière globale, donc on ne le renvoie pas.

1.1 Expressions

Les expressions regroupent tout ce qui a été demandé dans le sujet. Leur type est :

```
type expr_f =
  | Var    of var_f                        (* Feuille : variable *)
  | Bang   of expr_f                       (* Le déréférençage *)
  | Cst    of int                          (* Feuille : constante *)
  | Bin    of expr_f * operator_f * expr_f (* opérations binaires *)
  | PrInt  of expr_f
  | Let    of pattern_f * expr_f * expr_f      (* let <var_f> = <expr_f> in <exec_f>   *)
  | LetRec of var_f * expr_f * expr_f      (* let rec *)
  | If     of bexpr_f * expr_f
  | IfElse of bexpr_f * expr_f * expr_f
  | Fun    of pattern_f * expr_f               (* car les fonctions sont un objet fun var -> expr *)
  | App    of expr_f * expr_f              (* Ce sont les applications *)
  | Aff    of expr_f * expr_f               (* Affectation i.e le ':=' *)
  | Alloc  of expr_f                       (* Allocation mémoire *)
  | Pair   of expr_f * expr_f
  | Unit
```

Voici un tableau de correspondance entre les constructeurs et les notions OCamL.

Constructeur	Équivalent OCamL
`Var of var_f`	`x, y, c0, variable_1` : nom de variable
`Bang of expr_f`	`!` : déréférençage
`Cst of int`	`0`, `1` : les entiers
`Bin of expr_f * operator_f * expr_f`	`x+5` : opérations binaires (`+, -, *, /, mod`)
`PrInt of expr_f`	`let prInt x = print_int x ; print_newline ()`
`Let of pattern_f * expr_f * expr_f`	`let ... = ... in ...`
`LetRec of var_f * expr_f * expr_f`	`let rec ... = ... in ...`
`If of bexpr_f * expr_f`	`if ... then ... else 0`
`IfElse of bexpr_f * expr_f * expr_f`	`if ... then ... else ...`
`Fun of pattern_f * expr_f`	`fun x -> ...`
`App of expr_f * expr_f`	`a b` : application
`Aff of var_f * expr_f`	`... := ...`
`Alloc of expr_f`	`ref ...` : allocation mémoire
`Pair of expr_f * expr_f`	`... , ...` : couples

Certains constructeurs intermédiaires sont détaillés dans `type.ml`.

1.2 Valeurs

Les programmes fouine sont des expressions `expr_f`, et sont évaluées par la fonction `eval`. Une évaluation renvoie une valeur du type donné ci-dessous :

```
type val_f =
    Unit
  | Int  of int
  | Addr of Int32
  | Fun  of var_f * expr_f * env_f
  | Pair of val_f * val_f
 ;;
```

Concernant les fonctions, celles-ci sont de la forme `fun x -> expr`. On sauvegarde de plus l'environnement dans lequel elles ont été définies (notion de clotûre). Nous y reviendrons.

1.3 Environnement

L'environnement est une liste d'association (variable, valeur), qui agit comme une **pile** (on empile les associations les unes après les autres). Une variable est simplement une chaîne de caractères. Ainsi :

```
type env_f = (var_f * val_f) list
```

Si `x` est une variable, sa valeur associée dans un environnement `env` est la première occurrence `("x", ...)` dans l'environnement. Donc, si on considère la liste `l = [("x", Int(0)) ; ("x", Int(5))]`, la valeur de `x` courante est `Int(0)`.

Si on essaie de lire la valeur d'une variable non-existante, l'interpréteur lève une exception.

1.4 Mémoire

La mémoire est un dictionnaire d'association `Int32 -> val_f`. On définit `mem_f` comme le module `Mem_f` :

```
module Mem_f = Map.make(Int32)
(* ici, on est obligé de prendre des 'Int32' pour faire les adressages *)
(* car 'Map.make' doit prendre un module en argument *)
```

C'est la mémoire qui gère les références. Pour les gérer, on descend assez bas-niveau dans la philosophie. Les références seront vues comme une adresse mémoire. Ainsi, `let a = ref 0` est interprété comme

— Allouer une nouvelle case mémoire
— Associer `a` à cette nouvelle case *(notion d'adresse mémoire)*
— Mettre le contenu de cette case à 0

Pour savoir qu'elle est la prochaine case mémoire allouable, on utilise une variable globale `available`.

2 Fonctions

2.1 Fonctions classiques

Les fonctions sont définies avec une clôture pour la raison suivante

```
let a = 5 in
let f = fun x -> a in
let a = 10 in
f 10;;
```

Ici, `f` doit se souvenir de la valeur de `a`. Sauvegarder une clôture est en $\mathcal{O}(1)$: il s'agit juste de sauvegarder un pointeur vers une tête de pile i.e l'environnement courant. La valeur d'une fonction est

```
| Fun_var of var_f * expr_f * env_f
```

donc les fonctions ne sont pas typées en fouine.

2.2 Fonctions Récursives

Quand on définit une fonction `f` de la manière ci-dessus, il faut remarquer qu'au moment de sa définition, `f` n'est pas définie dans l'environnement, donc une fonction standard n'est pas définie dans sa propre clôture. On ne peut donc pas définir des fonctions récursives. On introduit le `let rec`.

Le `let rec` est un peu particulier. Il regarde si on est en train de définir une fonction ou non. Si oui, il construit la clôture de `f` en y rajoutant l'association `(f, clôture où f est défini)`. Il y a donc une définition cyclique ici, qui fera apparaître `<cycle>`. Je renvoie au code pour l'implémentation de ça.

Si c'est une variable quelconque, qui n'est pas une fonction, on se comporte comme un `let` classique. Ceci ne pose pas trop de problèmes si on ne joue qu'avec des `int` ou des couples, mais fait tout de même apparaître quelques petits cas pathologiques (cf dernière partie).

3 Parsing

3.1 `let ... in`

Ce qui pose problème avec les `let ... in ...` c'est qu'il y en a beaucoup de variantes ! On distingue les `let` des expressions des `let` extérieurs (du toplevel) qui permettent une syntaxe un peu particulière pour

enchaîner les `let` sans `in` ni `; ;`. Il y a aussi l'ajout (parfois facultatif) du mot clé `rec` qui dédouble tous les cas. Enfin les `let` peuvent servir à définir des fonctions ou des couples (triplets, etc.). Il y avait sûrement un moyen de synthétiser tous ces cas mais expliciter tous les cas étaient plus simple.

3.2 L'application de fonction

Typiquement, une application de fonctions est du type expression expression, mais écrire cela en tant que règle aurait abouti à du "rule never reduce" car on peut toujours lire une expression derrière. Là encore, le plus simple était d'expliciter tous les cas dans une règle "applicator applicated".

Pour la définition de fonction, il a fallu faire attention aux différentes façons de définir une fonction (avec des `fun ->` ou directement).

4 Pathologies

Dans cette partie, quelques pathologiques sur le fonctionnement de fouine.

4.1 Types `unit -> 'a`

Le type `unit -> 'a` existe en OCamL par exemple avec

```
let f () = 5 ;;
```

Ce code ne va pas parser en fouine car `()` ne correspond pas à un pattern de variables, or le constructeur pour les fonctions est `Fun of pattern_f * expr_f`. On utilise la variable "anonyme" `_` pour cela :

```
let f _  = 5 ;;
```

Ceci aura le comportement souhaité en pratique, car fouine ne vérifie pas les typages, et toutes les associations `_ <- ...` sont ignorées par l'environnement. Ainsi, le type `unit -> 'a` devient `'a -> 'b`.

4.2 Fonctions récursives

On remarquera simplement que le code

```
let f = 5 in
let rec f = f;;
```

va être accepté dans notre *Fouine*, alors qui n'est pas sensé l'être en OCamL. Par contre, le code suivant ne l'est pas :

```
let rec f = fun x -> f;;
```