

Fouine

Victor Boone - Gabrielle Pauvert

Mars 2018

Table des matières

1	Généralités	1
1.1	Expressions	2
1.2	Valeurs	3
1.3	Environnement	3
1.4	Mémoire	4
2	Parsing	4
2.1	<code>let ... in</code>	4
2.2	L'application de fonction	4
3	Evaluation	4
3.1	Style d'évaluation	4
3.2	Fonctions	5
3.2.1	Fonctions classiques	5
3.2.2	Fonctions Récursives	5
4	Exceptions	5
5	Transformations de Programmes	7
5.1	Impératives	7
5.2	Continuations	8
6	La machine SECD	9
6.1	La transformation	9
6.2	L'exécution	10
7	Pathologies	10
7.1	Types <code>unit -> 'a</code>	10
7.2	Fonctions récursives	11
7.3	Références	11

1 Généralités

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

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

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 -> (val_f -> val_f) -> (val_f -> val_f) list -> val_f
```

utilisée de la forme `eval expr env k kE`, et implémentée par continuations. La mémoire `mem_f` est globale.

1.1 Expressions

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

```
type expr_f =
| Cst      of int                (* Feuille : constante *)
| Bool     of bool              (* Feuille : booléen *)
| Var      of var_f             (* Feuille : variable *)
| Bang     of expr_f            (* Feuille : le déréférencage *)
| Unit     of unit              (* Feuille : le type unit *)
| Pair     of expr_f * expr_f   (* Un couple d'expressions *)
| Neg      of expr_f            (* Négation de Booléens *)
| Bin      of expr_f * operator_f * expr_f (* opérations binaires *)
| PrInt    of expr_f           (* built-in prInt *)
| 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 *)
| Match    of expr_f * pmatch_f (* match [expr_f] with [pattern_matching] *)
| IfElse   of expr_f * expr_f * expr_f (* If .. then .. else *)
| 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 *)
| Try      of expr_f * var_f * expr_f (* Le 'try ... with E ... -> g...' *)
| Raise    of expr_f           (* raise E ... : qui sera un int en pratique *)
```

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	! : déréférencage
Bool of bool	true, false : les booléens
Cst of int	0, 1 : les entiers
Neg of expr_f	not ... : négation booléenne
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 ...
IfElse of expr_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 expr_f * expr_f	... := ...
Alloc of expr_f	ref ... : allocation mémoire
Pair of expr_f * expr_f	... , ... : couples
Raise of expr_f	raise (E ...) : levée d'exceptions E
Try of expr_f * var_f * expr_f	try ... with E x -> ...

Certains constructeurs intermédiaires sont détaillés dans `type.ml`. Les booléens font leur apparition dans les expressions de ce *Fouine*, principalement pour simplifier les transformations de programmes. `Bin` a été étendu en conséquence.

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
  | Bool      of bool
  | Int       of int
  | Ref       of int
  | Cons      of string * val_f
  | Pair_val  of val_f * val_f
  | Fun_val   of pattern_f * expr_f * env_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. `Cons` n'est pas utilisé.

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 `Failure`.

1.4 Mémoire

La mémoire est un tableau de valeurs de taille fixe 1000000. Ainsi, on impose à *Fouine* une quantité bornée de mémoire. Si celle-ci est dépassée, l'interpréteur lève **Failure "Out of Memory"**. Les cases mémoires étant typées `val_f`, elles peuvent occuper une taille non bornée de mémoire.

```
let mem = Array.make 1000000 Unit
let available = ref 0;;
```

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 Parsing

2.1 `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.). Les `let` ont été un peu regroupés depuis le rendu 3. Une variable simple est vue comme un cas particulier de uplet à un élément. On définit une règle `var_pattern` qui admet tous ces cas. Les `let rec` ne sont pas concernés : on ne peut définir qu'au plus une variable avec un `let rec`.

2.2 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, et choisir de `shift`. Le plus simple était d'explicitement tous les cas dans une règle `applicator applicated` où `applicator` sert à enchaîner les applications (curryfication) et `applicated` détaille les possibilités de fonctions et d'arguments.

Pour la définition de fonction, il a fallu faire attention aux différentes façons de définir une fonction (avec des `fun ->` ou directement, ce qui complique encore un peu les définitions de `let... in...`).

3 Evaluation

3.1 Style d'évaluation

L'évaluation se fait par continuation : `eval expr env k kE`. La fonction d'évaluation reçoit quatre arguments :

- `expr` l'expression à évaluer
- `env` l'environnement courant (définition des variables)
- `k` la continuation courante
- `kE` la pile de continuations d'exceptions

Nous renvoyons au code pour l'implémentation de `eval`. Certains passages du code sont plus lourd car `eval` est capable d'afficher l'expression courante en cas d'erreur - l'implémentation de l'évaluation fainéante a aussi été gourmande en lignes de code.

3.2 Fonctions

3.2.1 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.

3.2.2 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>`. Un détail important ; lors d'un

```
let rec f = e1 in e2
```

on commence par évaluer `e1`. Il y a alors deux possibilités. Soit on reçoit une valeur fonctionnelle `Fun_val("f", e0, env0)`, avec `env0` une clôture dans laquelle `f` n'est pas défini. C'est là qu'on utilise le `let rec` d'OCaml pour redéfinir la clôture de la fonction avant de rajouter `f` dans l'environnement. On évalue ensuite `e2`.

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).

4 Exceptions

`eval` est implémenté par continuations, et dispose d'une pile de continuations d'exceptions. En cas de `try e1 with e2` l'interpréteur ajoute une nouvelle continuation d'exception sur la pile pour l'évaluation de `e1`. Quant à `raise e1`, il évalue `e1`, puis dépile la pile de continuations d'exceptions pour traiter le résultat.

Remarque : On pourrait se débarrasser de la pile, pour n'avoir qu'une continuation d'exception à chaque fois en changeant

```

| Raise expr ->
    begin
        match k' with
        | [] -> failwith "Raise : Nothing to catch exception"
        | k_exn :: k' -> eval expr env k_exn k'
    end
| Try (expr1, var_except, expr2) ->
    eval expr1 env k ((fun exn -> eval expr2 (env_aff var_except exn env) k k') :: k')

en

| Raise expr ->
    eval expr1 env k' k'
| Try (expr1, var_except, expr2) ->
    eval expr1 env k (fun exn -> eval expr2 (env_aff var_except exn env) k k')

```

Ceci nous a surpris à première vue, mais c'est ce qu'on a fait pour les transformations de programme et cela semble très bien marcher.

5 Transformations de Programmes

On met ici les formules utilisées pour les transformations de programmes. Ces dernières ont été converties en expressions d'arbres de programme en utilisant directement le *parser*.

5.1 Impératives

On rajoute en pratique un `__` aux variables utilisées pour différencier de celles du programme initial.

```
[| 42 |] (* devient *) fun s -> (42,s)
[| true |] (* devient *) fun s -> (true,s)
[| x |] (* devient *) fun s -> (x,s)
[| !e |] (* devient *) fun s -> let (l,s1) = [| e |] s in
    let v = read s1 l in (v,s1)
[| () |] (* devient *) fun s -> ((),s)

[| (e1, e2) |] (* devient *)
    fun s -> let (v2,s2) = [| e2 |] s in
        let (v1,s1) = [| e1 |] s2 in
            ((v1,v2),s1)

[| not e0 |] (* devient *)
    fun s -> let (b,s0) = [| e0 |] s in
        (not b, s0)
[| e1 + e2 |] (* devient *)
    fun s -> let (v2,s2) = [| e2 |] s in
        let (v1,s1) = [| e1 |] s2 in
            (v1 op v2,s1)

[| prInt e0 |] (* devient *)
    fun s -> let (v0,s0) = [| e0 |] s in
        (prInt v0, s0)
[| let pat = e1 in e2 |] (* devient *)
    fun s -> let (pat,s1) = [| e1 |] s in
        [| e2 |] s1
[| let rec v = e1 in e2 |] (* devient *)
    fun s -> let rec v = (let (f,s0) = [| e1 |] s in f)
        in [| e2 |] s

[| match |] (* non supporté *)
[| if b then e1 else e2 |] (* devient *)
    fun s -> let (b0, s0) = [| b |] s in
        if b0 then [| e1 |] s0
        else [| e2 |] s0

[| fun pat -> e |] (* devient *) fun s -> ((fun pat -> [| e |]), s)
[| e1 e2 |] (* devient *)
    fun s -> let (v, s2) = [| e2 |] s in
        let (f, s1) = [| e1 |] s2 in
            f v s1

[| e1 := e2 |] (* devient *)
```

```

fun s -> let (l1, s1) = [| e1 |] s in
        let (v2, s2) = [| e2 |] s2 in
        let s3 = write s2 l1 v2 in
        ((), s3)
[| ref e0 |] (* devient *)
fun s -> let (v,s1) = [| e0 |] s in
        let (s2,l) = alloc s1 in
        let s3 = write s2 l v in
        (l,s3)

[| try e1 with E x -> e2 |] (* devient *)
fun s -> try let (v1, s1) = [| e1 |] s in (v1, s1)
        with E x -> [| e2 |] s

[| raise e0 |] (* devient *)
fun s -> try let (v,s0) = [| e0 |] s in
        (raise (E v), s0)

```

5.2 Continuations

On rajoute en pratique un `_` aux variables utilisées pour les différencier de celles du programme initial.

```

[| () |] (* devient *) fun k kE -> k ()
[| true |] (* devient *) fun k kE -> k true
[| 42 |] (* devient *) fun k kE -> k 42
[| x |] (* devient *) fun k kE -> k x

[| !e |] (* devient *)
fun k kE -> [| e |] (fun addr -> k (!addr)) kE

[| not e |] (* devient *)
fun k kE -> [| e |] (fun b -> k (not b)) kE
[| e1 + e2 |] (* devient *)
fun k kE -> [| e2 |] (fun v2 -> [| e1 |] (fun v1 -> k (v1 + v2)) kE) kE
[| prInt e |] (* devient *)
fun k kE -> [| e |] (fun v -> k (prInt v)) kE

[| if be then e1 else e2 |] (* devient *)
fun k kE -> [| be |] (fun b -> (if b then [| e1 |] else [| e2 |]) k kE) kE

```

C'est là qu'il est agréable de supporter les booléens, car `[| be |]` renvoie une valeur booléenne. Il est possible d'esquiver ce problème autrement, mais de manière moins fluide.

```

[| let x = e1 in e2 |] (* devient *)
fun k kE -> [| e1 |] (fun v -> let x = v in [| e2 |] k kE) kE
[| let rec f = e1 in e2 |] (* devient *)
fun k kE -> [| e1 |] (fun v -> let rec f = v in [| e2 |] k kE) kE

```

Ici, on utilise le fait que `[| e1 |]` renvoie une valeur fonctionnelle, mais non récursive. L'utilisation d'un `let rec` juste derrière permet de redéfinir cette valeur fonctionnelle comme étant récursive. Ceci fonctionne car *Fouine* n'évalue pas les fonctions, mais les renvoie directement avec leur clôture.


```

[| match |] (* non supporté *)
[| fun x -> e |] (* devient *)
    fun k kE -> k (fun x -> [| e |])
[| e1 e2 |] (* devient *)
    fun k kE -> [| e2 |] (fun v -> [| e1 |] (fun f -> f v k kE) kE) kE

[| e1 := e2 |] (* devient *)
    fun k kE -> [| e2 |] (fun v -> [| e1 |] (fun addr -> k (addr := v)) kE) kE
[| ref e |] (* devient *)
    fun k kE -> [| e |] (fun v -> k (ref v)) kE

[| (e1, e2) |] (* devient *)
    fun k kE -> [| e2 |] (fun v2 -> [| e1 |] (fun v1 -> k (v1, v2)) kE) kE

[| try e1 with E x -> e2 |] (* devient *)
    fun k kE -> [| e1 |] k (fun x -> [| e2 |] k kE)
[| raise e |] (* devient *)
    fun k kE -> [| e |] (fun v -> kE v) kE

```

6 La machine SECD

6.1 La transformation

Le code de la machine SECD est représenté par un tableau d'instructions, et ressemble beaucoup à de l'assembleur. Les indices du tableau sont utilisés pour effectuer des JUMP à des endroits précis du code (utile par exemple pour les conditions et les exceptions). On effectue la transformation en deux étapes : une première passe sert à déterminer le nombre d'instructions que l'on va avoir dans le tableau, ce qui nous permet de créer un array de la bonne taille, que l'on remplit avec les instructions pendant une deuxième passe.

Pour le cas du let : on prend en argument un uplet (var_pattern). A l'aide d'une fonction récursive auxiliaire traite_pattern, on ajoute autant de DESTRUCT puis de LET qu'il y a de variables dans cet uplet. On garde en mémoire le nombre de LET mis, puis on ajoute à la fin ce nombre de ENDLET dans le tableau d'instructions.

```

Cst a (* devient *) CONST a
Bool b (* devient *) BOOL b
Var x (* devient *) ACCESS x
!e (* devient *) [e]; READ
() (* devient *) UNIT
(e1,e2) (* devient *) [e2]; [e1]; PAIR
not(e) (* devient *) [e]; NOT
e1 + e2 (* devient *) [e1]; [e2]; ADD
(* de même pour les autres opérations arithmétiques et booléennes *)
(* cas particulier: or et and sont transformés pour l'évaluation fainéante:
a and b devient if a then b else false
a or b devient if a then true else b *)
prInt e (* devient *) [e]; PRINT
let pattern = e1 in e2 (* devient *) [e1]; LET (pattern); [e2]; ENDLET (pattern)
let rec f = e1 in e2 (* devient *) [e1]; REC f; [e2]; ENDLET
if b then e1 else e2 (* devient *) [b]; JUMPIF (addr1); [e2]; JUMP (addr2); [e1]
(* où addr1 pointe sur le début de [e1] et addr2 pointe sur la première instruction suivant [e1] *)
fun pattern -> e (* devient *) CLOSURE (addr); LET (pattern); [e]; ENDLET (pattern); RETURN

```

```

(* où addr pointe vers la première instruction suivant RETURN *)
e1 e2 (* devient *) [e2]; [e1]; APPLY
e1 := e2 (* devient *) [e2]; [e1]; WRITE
ref e (* devient *) [e]; ALLOC
try e with E x -> eX (* devient *) SETJMP (addr1); [e]; UNSETJMP; JUMP (addr2); LET x; [eX]; ENDLET
(* où addr1 pointe vers LET x et addr2 pointe vers la première instruction après le ENDLET *)
raise e (* devient *) [e]; LONGJMP

```

6.2 L'exécution

Mot clé	Env avant	Stack avant	PC avant	Env après	Stack après	PC après
CONST c	e	s	pc	e	c.s	pc+1
BOOL b	e	s	pc	e	b.s	pc+1
ACCESS x	e	s	pc	e	x.s	pc+1
ADD/etc.	e	v1.v2.s	pc	e	v1+v2.s	pc+1
NOT	e	b.s	pc	e	not(b).s	pc+1
EQ/etc.	e	v1.v2.s	pc	e	(v1 == v2).s	pc+1
PRINT	e	v.s	pc	e	v.s	pc+1
LET x	e	v.s	pc	(x,v).e	s	pc+1
REC x	e	v.s	pc	(x,v).e	s	pc+1
ENDLET	(x,v).e	s	pc	e	s	pc+1
JUMP addr	e	s	pc	e	s	addr
JUMPIF addr	e	b.s	pc	e	s	addr si b==True et pc+1 sinon
CLOSURE addr	?	?	?	?	?	?
APPLY	e	f.x.s	pc	e	(f x).s	pc+1
RETURN	e	addr.s	pc	e	s	addr
READ						
WRITE						
ALLOC						
SETJMP addr						
UNSETJMP						
LONGJMP						
PAIR						
DESTRUCT						

7 Pathologies

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

7.1 Types unit -> 'a

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

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

Ce code ne va pas passer 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.

7.2 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 (et c'est normal, il ne l'est pas non plus en Ocaml) :

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

7.3 Références

```
let a = ref (ref 42) in  
a := ref (ref 4);  
print !(!(a));;
```

Ici, on a un code fouine accepté, qui ne l'est pas en OCaml, car OCaml perçoit une erreur de typage.