Programmation Fonctionnelle: Evaluation des programmes `OCaml`

Yann Régis-Gianas
(IRIF, Univ. Paris Diderot, Inria) – `yrg@irif.fr`

2019-10-18

Sémantique d'`OCaml`

Sauriez-vous exécuter un programme `OCaml`
sans ordinateur?

Sémantique d'`OCaml`

Sauriez-vous exécuter un programme `OCaml` sans ordinateur?

Aujourd'hui, nous donnerons deux réponses distinctes à cette question:

- Une réponse "haut-niveau" : l'évaluation par substitution.
- Une réponse "bas-niveau" : l'évaluation comme en machine.

Evaluation par substitution

On peut expliquer les règles d'évaluation du sous-ensemble d'`OCaml` que nous avons étudié jusqu'à maintenant à l'aide d'un unique mécanisme : la **substitution**.

Pour le comprendre, nous avons besoin de la notion de **variable libre** et de **variable liée**.

Variable liée et occurrence libre

Variable liée

Dans une expression `OCaml`, une variable est dite **liée** si elle est introduite par les mot-clés **fun** ou **let** ou par un motif. Quand une variablie est liée, on peut remplacer son nom par un nom inutilisé sans changer ce que calcule l'expression. On dit aussi qu'une telle variable est muette.

Exemple: Dans "`fun x -> x`", "`let x = 0 in x`", ou encore "`function Some x -> x`", la variable `x` est toujours liée.

Variable libre

Dans une expression, une occurrence d'une variable `x` est **libre** si `x` n'est pas liée dans le contexte où cette occurrence apparaît.

Exemple: Dans "`fun y -> (x, fun x -> x)`" et "`let x = x in x`" la première occurrence de la variable `x` est libre, pas la seconde.

Substitution des occurrences libres

On note $t[x/u]$ pour la substitution par u des occurrences libres de x dans t.

Exemples:

- `(x + y)[x/1]` =

Substitution des occurrences libres

On note $t[x/u]$ pour la substitution par u des occurrences libres de x dans t.

Exemples:

- `(x + y)[x/1]` = `(1 + y)`
- `(let x = 0 in x + y)[x/1]` =

Substitution des occurrences libres

On note $t[x/u]$ pour la substitution par u des occurrences libres de x dans t.

Exemples:

- `(x + y)[x/1]` = `(1 + y)`
- `(let x = 0 in x + y)[x/1]` = `(let x = 0 in x + y)`
- `(fun x -> x)[x/1]` =

Substitution des occurrences libres

On note $t[x/u]$ pour la substitution par u des occurrences libres de x dans t.

Exemples:

- `(x + y)[x/1]` = `(1 + y)`
- `(let x = 0 in x + y)[x/1]` = `(let x = 0 in x + y)`
- `(fun x -> x)[x/1]` = `(fun x -> x)`
- `(x, fun x -> x)[x/1]` =

Substitution des occurrences libres

On note $t[x/u]$ pour la substitution par u des occurrences libres de x dans t.

Exemples:

- `(x + y)[x/1] = (1 + y)`
- `(let x = 0 in x + y)[x/1] = (let x = 0 in x + y)`
- `(fun x -> x)[x/1] = (fun x -> x)`
- `(x, fun x -> x)[x/1] = (1, fun x -> x)`
- `(let x = x in x + 1)[x/1] =`

Substitution des occurrences libres

On note $t[x/u]$ pour la substitution par u des occurrences libres de x dans t.

Exemples:

- `(x + y)[x/1] = (1 + y)`
- `(let x = 0 in x + y)[x/1] = (let x = 0 in x + y)`
- `(fun x -> x)[x/1] = (fun x -> x)`
- `(x, fun x -> x)[x/1] = (1, fun x -> x)`
- `(let x = x in x + 1)[x/1] = (let x = 1 in x + 1)`
- `(fun y -> x)[x/x + 1] =`

Substitution des occurrences libres

On note $t[x/u]$ pour la substitution par u des occurrences libres de x dans t.

Exemples:

- `(x + y)[x/1] = (1 + y)`
- `(let x = 0 in x + y)[x/1] = (let x = 0 in x + y)`
- `(fun x -> x)[x/1] = (fun x -> x)`
- `(x, fun x -> x)[x/1] = (1, fun x -> x)`
- `(let x = x in x + 1)[x/1] = (let x = 1 in x + 1)`
- `(fun y -> x)[x/x + 1] = (fun y -> x + 1)`
- `(fun y -> x)[x/y + 1] =`

Substitution des occurrences libres

On note $t[x/u]$ pour la substitution par u des occurrences libres de x dans t.

Exemples:

- `(x + y)[x/1] = (1 + y)`
- `(let x = 0 in x + y)[x/1] = (let x = 0 in x + y)`
- `(fun x -> x)[x/1] = (fun x -> x)`
- `(x, fun x -> x)[x/1] = (1, fun x -> x)`
- `(let x = x in x + 1)[x/1] = (let x = 1 in x + 1)`
- `(fun y -> x)[x/x + 1] = (fun y -> x + 1)`
- `(fun y -> x)[x/y + 1] = (fun z -> y + 1)`

Observez le dernier exemple:
Pourquoi faut-il renommer la variable muette de la fonction? Pourquoi `(fun y -> y + 1)` n'est pas une bonne réponse?

Evaluation des expressions `OCaml`

Evaluer une expression, c'est lui appliquer des règles de calcul tant que l'on n'obtient pas une valeur. Une valeur est par définition un expression valide que l'on ne plus réduire plus, c'est le résultat du calcul.

Quelles sont les valeurs du fragment fonctionnel d'`OCaml`?

- Les littéraux (entiers, booléens, chaînes de caractères, caractères, flottants, ...).
- Les fonctions de la forme `fun x_1 ... x_n -> e` qui n'a pas d'occurrence de variable libre.
- Les enregistrements de la forme `{ l_1 = v_1 ; ...; l_n = v_n }`, c'est-à-dire les enregistrements dont les champs sont des valeurs.
- Les valeurs étiquetées de la forme `C (v_1, ..., v_n)` où `C` est une constructeur de données.

Comment évaluer le `let`?

Soient `e_1` et `e_2` deux expressions.

L'expression "`let x = e_1 in e_2`" n'est pas une valeur alors comment l'évaluer?

1. Évaluer `e_1` en sa valeur `v_1` (si le calcul termine).
2. Évaluer l'expression `e_2[x / v_1]`.

Exemples:

```
1   let x = 1 + 2 in 2 * x
2   (* → *) let x = 3 in 2 * x
3   (* → *) 2 * 3
4   (* → *) 6
5
6   let x = 1 + 2 in let x = 2 * x in 3 * x
7   (* → *) let x = 3 in let x = 2 * x in 3 * x
8   (* → *) let x = 2 * 3 in 3 * x
9   (* → *) let x = 6 in 3 * x
10  (* → *) 3 * 6
11  (* → *) 18
```

Comment évaluer l'application?

Soient `e_1` et `e_2` deux expressions.

L'expression "`e_1 e_2`" n'est pas une valeur alors comment l'évaluer?

1. Evaluer `e_1` : on doit obtenir une fonction de la forme `fun x -> e` (si le calcul termine).
2. Evaluer `e_2` et obtenir une valeur `v_2` (si le calcul termine).
3. Evaluer `e[x / v_2]`.

Exemples:

```
(let y = 2 in fun x -> y + x) (1 + 2)
(* → *) (fun x -> 2 + x) (1 + 2)
(* → *) (fun x -> 2 + x) 3
(* → *) 2 + 3
(* → *) 5

(fun x -> fun y -> y + x) (1 + 2)
(* → *) (fun x -> fun y -> y + x) 3
(* → *) (fun y -> y + 3)
```

Comment évaluer le `match`?

L'expression "`match e with p_1 -> e_1 | ... | p_n -> e_n`" n'est pas une valeur alors comment l'évaluer?

1. Evaluer `e` en une valeur `v` (si le calcul termine).
2. Essayer de filter `v` par le motif `p_1` :
   - 2.1 Le filtrage réussi produit une liste de substitutions de la forme `[x_i / v_i]` où `x_i` est une variable du motif et `v_i` est une valeur extraite de `v`. On applique ces substitution sur `e_1` et on évalue l'expression résultante.
   - 2.2 Le filtrage a échoué : on passe à la branche suivante et ainsi de suite. L'exhausitivité des analyses nous assure qu'une des analyses sera couronnée de succès!

Exemples:

```
1  let x = S (42, true) in match x with N -> 0 | S (y, x) -> if x then y else -y
2  (* → *) match S (42, true) with N -> 0 | S (y, x) -> if x then y else -y
3  (* → *) match S (42, true) with S (y, x) -> if x then y else -y
4  (* → *) if true then 42 else -42
5  (* → *) 42
```

Comment évaluer le `let rec`?

L'expression "`let rec x = e_1 in e_2`" n'est pas une valeur alors comment l'évaluer?

1. Evaluer `e_1` en une valeur `v_1` (si le calcul termine). Cette valeur contient des occurrences libres de `x`.
2. Construire la valeur cyclique : `v = v_1[x/v]`
3. Evaluer `e_2[x/v]`.

Exemples:

```
1  let rec f = fun x -> if x = 0 then 1 else f (x - 1) in f 1
2  (* → *) v 1
3  (* où on a construit *) v = fun x -> if x = 0 then 1 else v (x - 1)
4  (* → *) if 1 = 0 then 1 else v (1 - 1)
5  (* → *) v (1 - 1)
6  (* → *) v 0
7  (* → *) if 0 = 0 then 1 else v (0 - 1)
8  (* → *) 1
```

Et pour les autres constructions?

C'est facile!

- Pour évaluer une expression de la forme `{l_1 = e_1; ...; l_n = e_n}`, on évalue chacune des `e_i`.
- Pour évaluer une expression de la forme `C (e_1, ..., e_n)`, on évalue chacune des `e_i`.

D'accord, mais ça se passe comme ça dans la machine?

Les règles que nous venons de voir sont correctes mais ne correspondent pas à ce qui se passent vraiment dans l'ordinateur lorsque l'on évalue un programme `OCaml`.

En effet, le mécanisme de substitution est difficile à implémenter efficacement. Par ailleurs, construire de nouvelles fonctions en appliquant une substitution sur le code machine d'une autre fonction sera très délicat à implémenter.

On utilise plutôt un mécanisme dit d'**évaluation sous environnement** et aussi une représentation de plus bas-niveau des valeurs.

Comment sont représentées les valeurs `OCaml` en machine?

- Les entiers et les booléens sont représentés par des mots de 63 bits, les booléens, les chaînes de caractères par des blocs d'octets commençant par un entier qui code leur longueur, les caractères par des octets, les flottants en suivant le standard IEEE.
- Les enregistrements de la forme `{ l_1 = v_1 ; ...; l_n = v_n }` sont des blocs alloués dans le tas et formés des valeurs des champs.
- Les valeurs étiquetées de la forme `C (v_1, ..., v_n)` où `C` est une constructeur de données sont aussi des blocs alloués dans le tas et dont la première case contient un entier qui code le construceur `C`. En `OCaml`, il y a aussi le cas des constructeurs de données constants qui sont directement représentés par des entiers.

Comment sont représentées les valeurs `OCaml` en machine?

- Les entiers et les booléens sont représentés par des mots de 63 bits, les booléens, les chaînes de caractères par des blocs d'octets commençant par un entier qui code leur longueur, les caractères par des octets, les flottants en suivant le standard IEEE.
- Les enregistrements de la forme `{ l_1 = v_1 ; ...; l_n = v_n }` sont des blocs alloués dans le tas et formés des valeurs des champs.
- Les valeurs étiquetées de la forme `C (v_1, ..., v_n)` où `C` est une constructeur de données sont aussi des blocs alloués dans le tas et dont la première case contient un entier qui code le construceur `C`. En `OCaml`, il y a aussi le cas des constructeurs de données constants qui sont directement représentés par des entiers.
- **Et les fonctions?**

Comment sont représentées les fonctions en machine?

On veut compiler une fonction une fois pour toute le code d'une fonction et ne jamais avoir à le modifier pendant l'exécution. Pour cela, on représente les fonctions par des **fermetures**. Une fermeture est un bloc alloué dans le tas et qui débute par un pointeur de code vers le code de la fonction. Ce bloc contient aussi les valeurs des variables libres qui apparaissent dans la fonction. Ainsi évaluer l'expression suivante :

```
1  let f x =
2    let z = 2 * x in fun y -> x + z + y
3  in f 1
```

s'évalue en une fermeture dont le pointeur de code addresse la fonction suivante:

```
1  let g y env = env[1] + env[2] + y
```

et dont les variables libres valent 1 et 2. Cette fonction est donc représentée en mémoire par le bloc de trois cases alloué sur le tas:

```
1  [ &g ; 1; 2 ] (* Pointeur de code, valeur de x, valeur de z. *)
```

Quand on applique une fermeture sur une valeur `v`, on commence par extraire le pointeur de code et on l'appelle en passant `v` en premier argument et la fermeture elle-même en second argument.

Prêt(e)s à implémenter un évaluateur d'`OCaml` en `OCaml`?