Sémantique opérationnelle

Evaluations par nom et par valeur

Etude d'un exemple

```
# let sqr = fun x -> x*.x;;

# let rec puiss n x = match n with
                0 ->  1.
              |  n when (n mod 2) = 0 -> sqr (puiss (n/2) x)
              |  n                    -> x*.sqr (puiss (n/2) x);;
```

Question : comment se fait l'évaluation de l'expression

```
sqr (puiss (n/2) x)
```

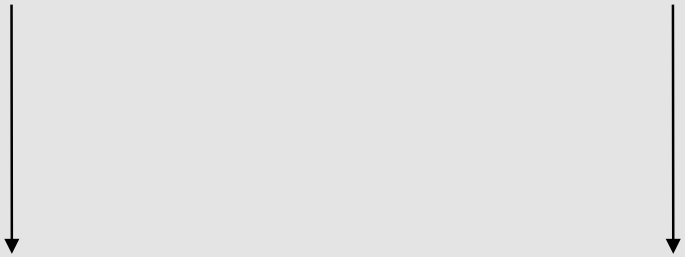
i.e. `(fun x -> x*.x) (puiss (n/2) x)`

Sémantique opérationnelle

Evaluations par nom et par valeur

Evaluation d'une application

(fun x -> x*.x) (puiss (n/2) x)



Fonction

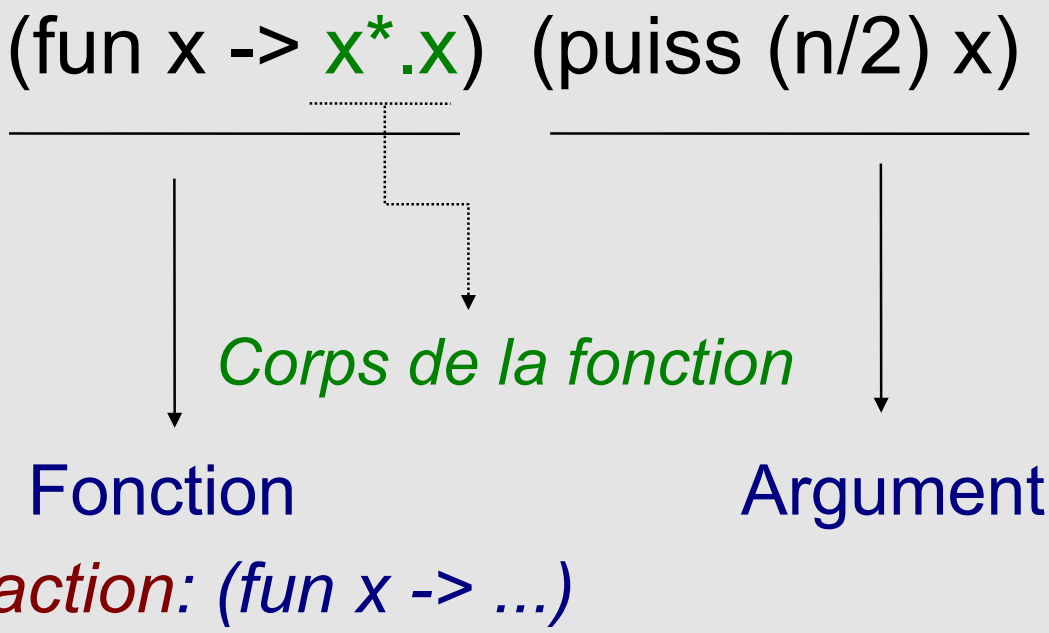
Argument

Abstraction: (fun x -> ...)

Sémantique opérationnelle

Evaluations par nom et par valeur

Evaluation d'une application



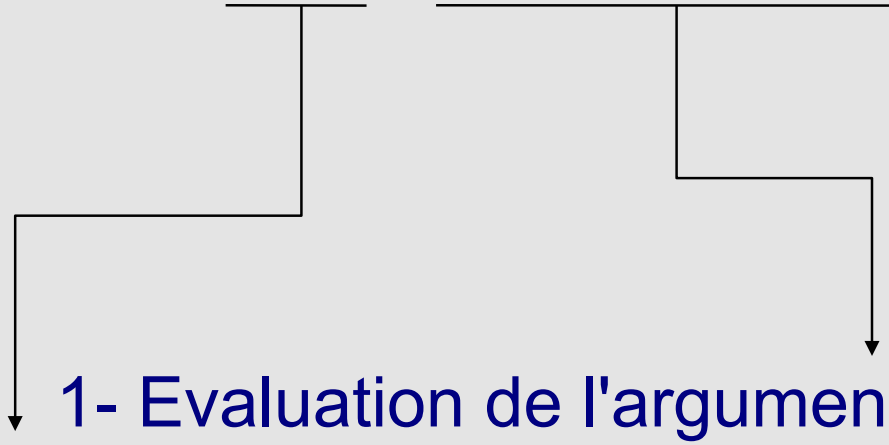
Abstraction: (fun x -> ...)

Sémantique opérationnelle

Evaluations par nom et par valeur

Evaluation par valeur

(fun x -> x*.x) (puiss (n/2) x)



1- Evaluation de l'argument : valeur v

2- Substitution de v à x dans le corps

3- Evaluation de v*.v

Dans le corps de la fonction,
la variable est remplacée par la valeur de l'argument

Sémantique opérationnelle

Evaluations par nom et par valeur

Evaluation par nom

(fun x -> x*.x) (puiss (n/2) x)

1- Substitution de (puiss (n/2) x) à x dans le corps

2- Evaluation de (puiss (n/2) x) *. (puiss (n/2) x)

Dans le corps de la fonction,
la variable est remplacée par le "nom" de l'argument avant évaluation de celui-ci

Sémantique opérationnelle

Evaluations par nom et par valeur

Evaluation par nom

(fun x -> x*.x) (puiss (n/2) x)

1- Substitution de (puiss (n/2) x) à x dans le corps

2- Evaluation de (puiss (n/2) x) *. (puiss (n/2) x)

Dans le corps de la fonction,
la variable est remplacée par le "nom" de l'argument avant évaluation de celui-ci

Quelle est la stratégie la plus efficace?

Sémantique opérationnelle

Evaluations par nom et par valeur

Evaluation par nom

(fun x -> x*.x) (puiss (n/2) x)

1- Substitution de (puiss (n/2) x) à x dans le corps

2- Evaluation de (puiss (n/2) x) *. (puiss (n/2) x)

Dans le corps de la fonction,
la variable est remplacée par le "nom" de l'argument avant évaluation de celui-ci

Quelle est la stratégie la plus efficace?
Les 2 stratégies donnent-elles le même résultat?

Sémantique opérationnelle

Evaluations par nom et par valeur

Les deux stratégies donnent-elles le même résultat?

```
# let c = fun x -> 0;;          val c : 'a -> int = <fun>
# let rec f n = n*(f n);;       val f : int -> int = <fun>
# c f;;
- : int = 0
# c (f 5);;
Stack overflow during evaluation (looping recursion?).
```

Comment expliquer ce dernier message?
Quelle est la stratégie d'évaluation de Caml?
Comment justifier ce choix?

Sémantique opérationnelle de Caml: *Evaluation par valeur*

Evaluation d'une expression sans variable libre

Constante ou abstraction

On ne fait rien! (expression déjà *réduite*)

```
# let c = fun x -> sin ( (3. *. 3.1416) /. 2.);;
val c : 'a -> float = <fun>
```

sin ((3. *. 3.1416) /. 2.) *n'a pas été calculé*

```
# c puiss;;
-   : float = -0.999999999939284

# c (puiss 64 1.);;
-   : float = -0.999999999939284
```

sin ((3. *. 3.1416) /. 2.) *est calculé à chaque appel (en théorie!)*

Sémantique opérationnelle de Caml: *Evaluation par valeur*

Evaluation d'une expression sans variable libre

Application (e_1 e_2)

- on évalue e_1 ---> v_1
- on évalue e_2 ---> v_2
- si v_1 est une abstraction : v_1 = *(fun x -> corps)*
  - on remplace *x* par v_2 dans *corps* et on obtient:

    corps [x <- v_2] *(renommage de variable)*
  - on réitère le processus sur corps [x <- v_2] jusqu'à une forme réduite

Sémantique opérationnelle de Caml: *Evaluation par valeur*

Exemple 1

```
let mult   = fun x y -> x*y;;
let triple = mult 3;;
let sqr    = fun x -> x*x;;
let comp   = fun f -> fun g -> fun x -> f (g x);;
```

Reduction de comp sqr triple (2*3)

i.e. ((comp sqr) triple) (2*3)

On devrait trouver (sqr ∘ triple) (6) = sqr (18)= 324

Sémantique opérationnelle de Caml: *Evaluation par valeur*

((comp sqr) triple) (2*3)

(comp sqr) triple

(comp sqr) = (fun f g x -> f (g x)) (fun x -> x*x)

(fun f -> (fun g x -> f (g x))) (fun x -> x*x) -->

(fun g x -> (fun x -> x*x) (g x)) =

(fun g x -> (fun z -> z*z) (g x))

ici renommage (facultatif) de variables liées: *α-conversion*

triple = mult 3 = (fun x y -> x*y) 3 =

(fun x -> (fun y -> x*y))) 3 -->

(fun y -> 3*y)

(comp sqr) triple -->

(fun g x -> (fun z -> z*z) (g x)) (fun y -> 3*y) -->

fun x -> (fun z -> z*z) ((fun y -> 3*y) x)

Sémantique opérationnelle de Caml: *Evaluation par valeur*

((comp sqr) triple) (2*3)

(comp sqr) triple →
(fun x -> (fun z -> z*z) ((fun y -> 3*y) x))

(2*3) --> 6

((comp spr) triple) (2*3) -->
(fun x -> (fun z -> z*z) ((fun y -> 3*y) x)) 6 -->
(fun z -> z*z) ((fun y -> 3*y) 6) =
(fun z -> z*z) ((fun y -> 3*y) 6)
(fun z -> z*z) (3*6) -->
(fun z -> z*z) (18) -->
18*18 --> 324

Sémantique opérationnelle
Retour sur l' η-réduction

Exemple 2

```
# let mult x y = x * y;;
# let triple1 = mult 3;;

triple1 = (mult 3) = (fun x y -> x*y) 3 =
     (fun x -> (fun y -> x*y)) 3  -->   (fun y -> 3*y)

# let triple2 = fun x -> mult 3 x;;
triple2 = fun x -> mult 3 x= fun x -> ((fun y z ->y*z) 3) x
```

triple2 reste tel quel: évaluation bloquée par le fun x -> ...

Sémantique opérationnelle
Retour sur l' η-réduction

```
# let triple1 = mult 3;;
# let triple2 = fun x -> mult 3 x;;
```

triple1 est l' *η-réduction* de triple2

triple2 est l' *η-expansion* de triple1

Même fonction mathématique

Mais opérationnellement différentes

```
triple1 =  (fun y -> 3*y)
triple2 =  fun x -> ((fun y z -> y*z)  3)  x
```

Leurs codes différent l'un de l'autre

Sémantique opérationnelle
Retour sur l' η-réduction

```
# let triple1 = mult 3;;
# let triple2 = fun x -> mult 3 x;;
```

triple1 est l' *η-réduction* de triple2

triple2 est l' *η-expansion* de triple1

Même *sémantique dénotationnelle*

Sémantique opérationnelle différente

```
triple1 =  (fun y -> 3*y)
triple2 =  fun x -> ((fun y z -> y*z)  3)  x
```

Leurs codes différent l'un de l'autre

Sémantique opérationnelle
Retour sur l' η-réduction

```
# let triple1 = mult 3;;
# let triple2 = fun x -> mult 3 x;;
```

triple1 est l' *η-réduction* de triple2

triple2 est l' *η-expansion* de triple1

Même *sémantique dénotationnelle*

Sémantique opérationnelle différente

```
triple1 =  (fun y -> 3*y)
```

immédiatement évalué

```
triple2 =  fun x -> ((fun y z -> y*z)  3)  x
```

évaluation retardée car c'est une abstraction

Sémantique opérationnelle
Application à la récursion

Point fixe d'une fonction d'ordre supérieur

let rec fact = *fun n -> if n=0 then 1 else n* fact(n-1)*

let sFact = fun f -> *fun n -> if n=0 then 1 else n* f (n-1)*

sFact sqr = *fun n -> if n=0 then 1 else n*sqr (n-1)*

sFact double = *fun n -> if n=0 then 1 else n*double (n-1)*

sFact fact = *fun n -> if n=0 then 1 else n*fact (n-1)*

sFact fact = fact

fact *point fixe* de sFact

Un “let rec” exprime une équation de point fixe

Sémantique opérationnelle
Application à la récursion

Point fixe d'une fonction d'ordre supérieur

let rec bidule = *expr(bidule)*;;

let s = fun f -> *expr(f)*;;

s bidule = *expr*(bidule) = bidule

bidule point fixe de s

Idée: définir un opérateur fix : s-> fix s tel que

s (fix s) = fix s

let rec bidule = *expr(bidule)*;; -- > let bidule = fix (fun f -> *expr(f));;*

Sémantique opérationnelle
Application à la récursion

Point fixe d'une fonction d'ordre supérieur

```
# let rec fix = fun s -> s(fix s);;
val fix : ('a -> 'a) -> 'a = <fun>

# let sFact = fun f-> fun  n -> if n=0 then 1 else n * f (n-1);;
val s : (int -> int) -> int -> int = <fun>

# let fact = fix sFact;;
Stack overflow during evaluation (looping recursion?).
```

Pourquoi?

Sémantique opérationnelle
Application à la récursion

fact = fix sFact

```
# let rec fix = fun s -> s(fix s);;
```

Evaluation de fact:

fix sFact = (fun s -> s (fix s)) sFact --> sFact (fix sFact)

L'évaluation boucle: on doit à nouveau évaluer (fix sFact)

car récursion sans cas terminal

Il faut bloquer l'évaluation de *(fix sFact)* en le mettant dans une abstraction (sous un “fun”)

Sémantique opérationnelle
Application à la récursion

fact = fix sFact

```
# let rec fix = fun s -> s(fix s);;
```

remplacé par

```
# let rec fix = fun s -> s (fun x -> fix s x);;
```

C'est une η-expansion de (fix s)

Sémantique opérationnelle
Application à la récursion

fact = fix sFact

```
# let rec fix = fun s -> s (fun x -> (fix s x));;
```

Evaluation de fact

```
fact = fix sFact = (fun s -> s (fun x -> (fix s x)) sFact  -->
        sFact (fun x -> fix sFact x)
```

Sémantique opérationnelle
Application à la récursion

fact = fix sFact

```
# let rec fix = fun s -> s (fun x -> (fix s x));;
```

Evaluation de fact

fact = fix sFact = (fun s -> s (fun x -> (fix s x)) sFact -->

sFact (fun x -> fix sFact x)

déjà réduit

a remplacé fix sFact*, qui provoquait la boucle*

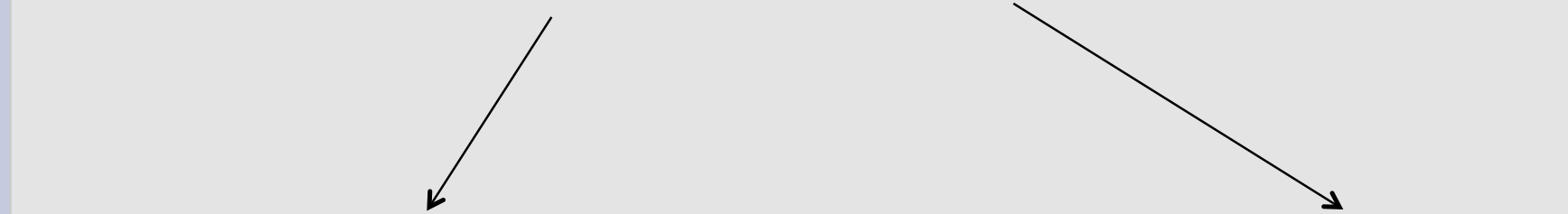
Sémantique opérationnelle
Application à la récursion

fact = fix sFact

```
# let rec fix = fun s -> s (fun x -> (fix s x));;
```

Evaluation de fact

```
fact = fix sFact = (fun s -> s (fun x -> (fix s x)) sFact  -->
                   sFact  (fun x -> fix sFact x) =
```



Sémantique opérationnelle
Application à la récursion

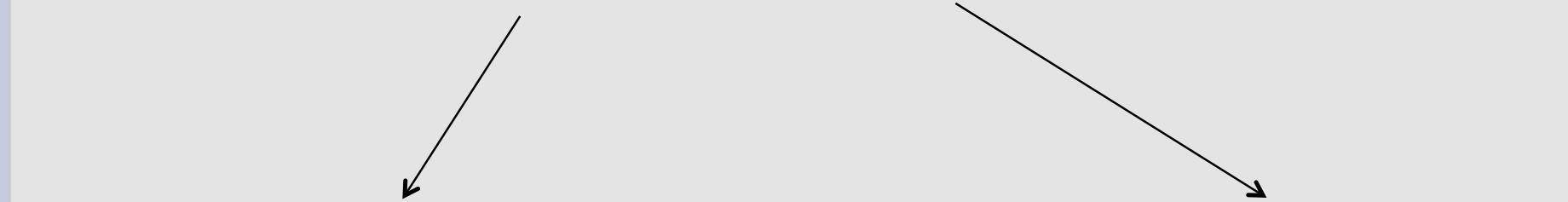
fact = fix sFact

```
# let rec fix = fun s -> s (fun x -> (fix s x));;
```

Evaluation de fact

fact = fix sFact = (fun s -> s (fun x -> (fix s x)) sFact -->

sFact (fun x -> fix sFact x) =



(fun f n -> if n=0 then 1 else n*f(n-1)) (fun x -> fact x) -->

fun n -> if n=0 then 1 else n*(fun x-> fact x)(n-1)

Sémantique opérationnelle
Application à la récursion

Evaluation de (fact n)

```
fact  -->   fun n -> if n=0 then 1 else n*(fun x -> fact x)(n-1)
```

Soit c une constante entière

- Si c=0

  (fact c) -- > 1

- Si c≠0

  (fact c) --> c*(fun x -> fact x)(c-1) -- > c*fact (c-1)

Sémantique opérationnelle
Application à la récursion

Evaluation de (fact n)

```
fact  -->   fun n -> if n=0 then 1 else n*(fun x -> fact x)(n-1)
```

Soit c une constante entière

- Si c=0

  (fact c) -- > 1

- Si c≠0

  (fact c) --> c*(fun x -> fact x)(c-1) -- > c*fact (c-1)

  ↓

  à nouveau évalué