Sémantique des langages

I] Présentation

But du cours : définir et étudier la *sémantique* **Question :** Comment définir la *signification* des programmes écrits dans un langage? La plupart du temps, on se contente d'une description informalle, en langage naturel. -> Ambigu, imprécis

La **sémantique formelle** définit mathématiquement le sens d'un langage. -> Utile si on veut écrire un compilateur ou un interprète ou si on veut prouver la correction d'un programme

Mais... C'est quoi un programme? -> Trop complexe à définir en tant qu'objet syntaxique (strings) -> On préfère la syntaxe abstraite C'est la façon dont sera représenté le programme, lors de la compilation, après l'analyse syntaxique

Par exemple, représentent le même arbre de syntaxe abstraite :

- `2*(x+1)`
- `(2 * ((x) + 1))`
- même chose mais avec des commentaires en plein milieu

On définit une syntaxe abstraite par une grammaire :

```
e :=  c   # constante
 | x      # variable
 | e + e  # addition
 | e * e  # multiplication
 | ...
```

Remarque : Dans cet exemple, `e1 + e2` est notée en empruntant le symbole de la syntaxe concrète. On aurait pu écrire `add(e1, e2)`, `+(e1, e2)`, etc.

En `OCaml`, on utilise les types construits :

```
type binop = Add | Mul | ...

type expression =
    | Cte of int
    | Var of string
    | Bin of binop * expression * expression
    | ...
```

Alors, `2*(x+1)` va être représenté par `Bin(Mul, Cte 2, Bin (Add, Var "x", Cte 1))`

Remarque : Ici, la grammaire est **non ambigue**, ie. pas besoin de parenthèses dans l'arbre de syntaxe abstraite.

On peut aussi avoir des construction propres à la syntaxe concrète -> **Syntaxic sugar** Par exemple, en `OCaml` : `[0, 1, 2]` pour `0 :: 1 :: 2 :: []`

C'est sur la syntaxe abstraite que l'on va définir la sémantique **Plusieurs types :**

- sémantique *axiomatique*
- sémantique *dénotationnelle*
- sémantique *par traduction*
- sémantique *opérationnelle*

1) Sémantique axiomatique

-> *logique de Hoare* (Tony Hoare, *An axiomatic basis for computer programming*, 1969) On introduit le triplet $\{P\}\ i\ \{Q\}$ signifiant : *"Si la formule P est vraie avant l'exécution de l'instruction i, alors la formule Q sera vraie après"*

Exemple : $\{x \geq 0\}\ x := x + 1\ \{x > 0\}$

Exemple de règle : $\{P[x \leftarrow E]\}x := E\{P(x)\}$

2) Syntaxe dénotationnelle

Elle associe à chaque expression e sa dénotation $[|e|]$, qui est un objet mathématique représentant le calcul désigné par e. Exemple : expressions arithmétiques : `e ::= x | n | e + e | e * e` Dénotation : $[|x|] = x \mapsto x$ $[|n|] = x \mapsto n$ $[|e1 + e2|] = x \mapsto [|e1|](x) + [|e2|](x)$ $[|e1 * e2|] = x \mapsto [|e1|](x) \times [|e2|](x)$

3) Sémantique par traduction

Elle consiste en la définition de la sémantique d"un langage en le **traduisant** vers un langage dont la sémantique est *déjà connue.*

4) Sémantique opérationnelle

Elle consiste à décrire l'**enchaînement des calculs** faits lors de l'*évaluation* d'un programme On peut le faire :

- **À grands pas -> sémantique naturelle**. On dit “cette expression d’évalue et donne cette valeur” : $e \twoheadrightarrow v$
- **À petits pas -> sémantique à réductions**. On dit “Ce programme se transforme, en un petit pas, en ce calcul et récurre” : $e \to e1 \to e2 \to \dots \to v$

II] Mini-ML

Syntaxe opérationnelle :

```
e ::=
    | x           # identificateur
    | c           # constante
    | op          # primitive (+, *, fst, ...)
    | fun x -> e      # fonction anonyme
    | e e             # application
    | (e, e)      # paire
    | let x = e in e  # liaison locale
```

Il manque des *conditionnelles*, malheureusement. Mais ce n’est pas un problème, car on peut rajouter ça comme **primitive**! `if e1 then e2 else e3` peut-être déclarée comme `opif(e1, (fun _ -> e2, fun _ -> e3))` On met `e2` et `e3` dans des fonctions pour *ne pas les évaluer* avant d’entrer dans le `if`!

De même, l’opérateur de **point fixe récursif**, `rec`, peut être défini comme `primitive`

On cherche à définir la sémantique opérationnelle du **Mini-ML** Les valeurs sont :

```
v ::=
    | c
    | op
    | fun x -> e
    | (v, v)
```

Pour définir $e \twoheadrightarrow v$, on a besoin de **règles d’inférence et de substitution**

Règle d’inférence: Une **relation** peut être définie comme la *plus petite relation* satisfaisant un ensemble d’**axiomes** sour la forme $\frac{}{P}$ et un ensemble d’**implication** sous la forme $\frac{P_1 P_2 \dots P_n}{P}$

Explication : ce qui est *sur* la barre est l’**hypothèse**, *au-dessous* la **conclusion**

Par exemple, on défniit $Pair(n)$ par :

$$\frac{}{Pair(0)} \text{ et } \frac{Pair(n)}{Pair(n+2)}$$

Arbre de dérivation : Application des *règles logiques* : les *nœuds* sont des **règles** et les *feuilles* des **axiomes**.

Par exemple :

$$\cfrac{\cfrac{\cfrac{}{Pair(0)}}{Pair(2)}}{Pair(4)}$$

L'ensemble des dérivations possibles caractérise exactement la plus petite relation satisfaisant les règles d'inférence.

Règle de substitution : L'ensemble des **variables libres** d'une exp `e`, notée `fv(e)` est défini par récurrence sur e :

- $fv(x) = \{x\}$
- $fv(c) = \emptyset$
- $fv(op) = \emptyset$
- $fv(fun\ x \rightarrow e) = fv(e) \setminus \{x\}$
- $fv(e1\ e1) = fv(e1) \cup fv(e2)$
- $fv((e2, e2)) = fv(e1) \cup fv(e2)$
- $fv(let\ x = e1\ in\ e2) = fv(e1) \cup (fv(e2) \setminus x)$

Une expression *sans variable libre* est dite **close**.

Définition : Substitution : Si e expression, x variable, v valeur, on note e[x <- v] la substitution de toute occurence libre de x par v dans e:

- x[x <- v] = v
- y[x <- v] = y si y $\neq$ x
- c[x <- v] = c
- op[x <- v] = op
- (fun x -> e)[x <- v] = fun x -> e
- (fun y -> e)[x <- v] = fun y -> e[x <- v] si y $\neq$ x
- (e1 e2)[x <- v] = (e1[x <- v] e2[x <- v])
- (e1, e2)[x <- v] = (e1[x <- v], e2[x <- v])
- (let x = e2 in e2)[x <- v] = let x = e1[x <- v] in e2
- (let y = e1 in e2)[x <- v] = let y = e1[x <- v] in e2[x <- v] si y $\neq$ x

On a donc les règles d'**inférence** du **Mini-ML** :

- $\dfrac{}{c \twoheadrightarrow c}$ # ie une constante s'évalue à elle-même
- $\dfrac{}{op \twoheadrightarrow op}$ # De même pour un opérateur
- $\dfrac{}{(fun\ x \to e) \twoheadrightarrow (fun\ x \to e)}$ # De même pour les onctions
- $\dfrac{(e_1 \twoheadrightarrow v_1).(e_2 \twoheadrightarrow v_2)}{(e_1, e_2) \twoheadrightarrow (v_1, v_2)}$ # Inférence aturelle sur les paires
- $\dfrac{(e_1 \twoheadrightarrow v_1).(e_2[x \leftarrow v_1]) \twoheadrightarrow v)}{(let\ x = e_1\ in\ e_2) \twoheadrightarrow v}$ # Inférence sur les liaisons locales
- $\dfrac{e_1 \twoheadrightarrow (fun\ x \to e) \quad e_2 \twoheadrightarrow v_2 \quad e[x \leftarrow v_2] \twoheadrightarrow v}{e_1\ e_2 \twoheadrightarrow v}$ # Inférence sur l'application de fonction

Note : on a ici un appel par valeur, ie. l'argument est totalement évalué avant l'appel.

Il faut rajouter des règles pour les primitives (e1 -> +).(e2 -> (n1, n2)).(n = n1 + n2)/(e1 e2 -> n) (e1 -> opif).(e2 -> (true, ((fun _ -> e3), (fun _ -> e4))).(e3 -> v)/(e1 e2 -> v) (e1 -> opfix).(e2 -> (fun f -> e)).(e[f <- opfix (fun f -> e)] -> v)/(e1 e2 -> v) (e1 -> fst).(e2 -> (v1, v2))/(e1 e2 -> v1)

III] Interprète

Remarque : dans notre sémantique, il existe des expression n'*ayant pas de valeur*. Par exemple `e = 1 2`

Pour établir une propriété d'une relation, on peut raisonner par **récurrence structurelle** sur la dérivation. Il faut traiter *tous les cas de dernière règle*.

Exemples: Proposition : Si e ->> v alors v est une valeur. De plus, si e est close, v l'est également **Preuve** par récurrence sur la dérivation (e ->> v)

Propositon (déterminisme de l'évaluation) Si $(e \twoheadrightarrow v)$ et $(e \twoheadrightarrow v')$ alors $v = v'$

On peut construire un **interprète**, donc une fonction en `OCaml` correspondant à la relation $\twoheadrightarrow$

```
type expression =
    | Var of string
    | Const of int
    | Op of string
    | Fun of string * expression
    | App of expression * expression
```

```
    | Pair of expression * expression
    | Let of string * expression * expression
;;

let rec subst e x v = match e with
    | Var y -> if y = x then v else e
    | Fun (y, e1) -> if y = x then e else Fun (y, subst e2 x v)
    | Let (y, e1, e2) -> Let (y, subst e1 x v, if y = x then e2 else subst e2 x v)
    | App (e1, e2) -> App (subst e1 x v, subst e2 x v)
    | Pair (e1, e2) -> Pair (subst e1 x v, subst e2 x v)
    | _ -> e
;;

let rec eval = function
    | Const _ | Op _ | Fun _ as v -> v
    | Pair (e1, e2) -> Pair (eval e1, eval e2)
    | Let (x, e1, e2) -> eval (subst e2 x (eval e1))
    | App (e1, e2) -> disjonction sur les primitives :))
```

Problèmes : - L'évaluation peut ne *pas faire de sens* et ne matcher sur rien - L'évaluation peut *boucler à l'infini*

Question : Peut-on **éviter** l'opération de *substitution*? **Idée :** utiliser un *dictionnaire* = **environnement**

Subtilité : Il faut se souvenir *où et quand* les variables sont substituées.

On utilise le module `Map`.

```
module Smap = Map.Make(String)
type value =
    | Vconst of int
    | Vop of string
    | Vpair of value * value
    | Vfun of string * environment * expression     # On a un environment pour chaque foncti
and environment = value Smap.t
```

Alors:

```
let rec eval env = function
    | Const n ->
        Vconst n
    | Op op ->
        Vop op
    | Pair (e1, e2) ->
        Vpair (eval env e1, eval env e2)
```

```
| Var x ->
    Smap.find x env
| Let (x, e1, e2) ->
    eval (Smap.add x (eval env e1)) e2
| Fun (x, e) ->
    Vfun (x, env, e)    # C'est la fermeture
| App (e1, e2) ->
    begin match eval env e1 with
        | Vfun (x, clos, e) ->
            eval (Smap.add x (eval env e2) clos) e
        | Vop "+" ->
            let Vpair (Vconst n1, Vconst n2) = eval env e2 in
                Vconst (n1 + n2)
        | ...
```

Inconvénients : la sémantique naturelle ne permet pas de distinguer les expressions dont le *calcul plante* de celles dont l'*évaluation ne termine pas*.

IV] Sémantique opérationnelle à petits pas

On remédie à ce problème en introduisant une notion d'**étape élémentaire** de calcul, itérée. On distingue alors 3 cas:

1) l'itération aboutit à une valeur $e_1 \rightarrow e_2 \rightarrow \cdots \rightarrow v$

2) l'itération bloque sur un irréductible qui n'est pas une valeur $e_1 \rightarrow e_2 \rightarrow ... \rightarrow e_n$

3) l'itération ne termine pas On commence par définir une relation $\stackrel{\epsilon}{\rightarrow}$ (réduction epsilon), réduction *"de tête"*, ie. au sommet de l'expression.

Deux règles : $(fun\ x \rightarrow e)v \stackrel{\epsilon}{\rightarrow} e[x \leftarrow v]$ $let\ x = v\ in\ e \stackrel{\epsilon}{\rightarrow} e[x \leftarrow v]$

N.B. : là encore, *appel par valeur*. On se donne aussi des règles pour les primitives, lorsqu'appliquées sur des valeurs.

Puis on se donne une **réduction en profondeur:** Règle d'inférence :

$$\frac{e_1 \stackrel{\epsilon}{\rightarrow} e_2}{E(e_1) \rightarrow E(e_2)}$$

Où E un **contexte**, défini par :

```
E ::=
  | $\square$
  | E e  (* réduction de la fonction *)
  | v E  (* une fois que la fonction est résolue, on résout l'argument *)
  | let x = E in e
  | (E, e)  (* Réduction du premier membre, avant le deuxième *)
  | (v, E)  (* On a le droit de réduire le second membre, mais que si le premier est aussi 
```

Les contextes définissent les *endroits du programme* où on a le droit de faire des calculs, ie. **ϵ-réduire**. Un contexte est dont un *"terme à trou"* où $\square$ représente le trou.

Exemple :

```
E ::= let x = +(2, $\square$) in let y = +(x, x) in y
```

$E(e)$ dénote le contexte dans lequel $\square$ a été remplacé par e.

La **règle d'inférence** permet d'évaluer, par **réduction en tête**, une sous-expression. Par définition, l'appel est *par valeur* et de la gauche vers la droite. Ainsi, ici, $(+(1, 2), \square)$ n'est pas un contexte vide!

On note $\rightarrow^\star$ la cloture réflexive et transitive de $\rightarrow$ On appelle **forme normale** toute *expression irréductible* $\{$valeurs$\}$ est dans $\{$formes normales$\}$ **/!\ pas d'égailté :** (1 2), par exemple, est irréductible mais n'est pas une valeur.

On va écrire

- `head_reduction : expression -> expression` # correspond à $\xrightarrow{\epsilon}$
- `decompose : expression -> context * expression` # décompose une exp sous la forme $E(e)$ avec e réductible en tête
- `reduce1 : expression -> expression option` # correspond à $\rightarrow$
- `reduce : expression -> expreeeion` # correspond à $\rightarrow^\star$

```
let rec is_a_value = function
    | Const _ | Op _ | Fun _ ->
      true
    | Var _ | App _ | Let _ ->
      false
    | Pair (e1, e2) ->
      is_a_value e1 && is_a_value e2
;;

let head_reduction = function
    | App (Fun, (x, e1), e2) when is_a_value e2 ->
      subst e1 x e2
```

```
    | Let (x, e1, e2) when is_a_value e1 ->
        subst e2 x e1
    | App *primitive*...
    | _ -> raise no_reduction
;;

type context = expression -> expression

let hole = fun e -> e
let app_left ctx e2 = fun e -> App (ctx e, e2).
let pair_left ctx e2 = fun e -> Pair (ctx e, e2)
let pair_right v1 ctx = fun e -> Pair (v1, ctx e)
let let_left x ctx e2 = fun e -> Let (x, ctx e, e2)

let rec decompose = function e ->
    (* Pas de décomposition possible *)
    | Var _ | Const _ | Op _ | Fun _ ->
        raise no_reduction
    (* Réduction en tête *)
    | App (Fun (x, e1), e2) when is_a_value e2 ->
        (hole, e)
    | Let (x, e1, e2) when is_a_value e1 ->
        (hole, e)
    | App *primitives*...
    (* Réduction en profondeur *)
    | App (e1, e2) ->
        if is_a_value e1 then
            let (ctx, rd) = decompose e2 in
            (app_right e1 ctx, rd)
        else
            let (ctx, rd) = decompose e1 in
            (app_left ctx e2, rd)
    | Let (x, e1, e2) ->
        let (ctx, rd) = decompose e1 in
        (let_left x ctx e2, rd)
    | Pair (e1, e2) ->
        ...

let reduce1 e = (* Fonction qui fait un pas de calcul *)
    try
        let ctx, e' = decompose e in
        Some (ctx (head_reduction e'))
    with no_reduction ->
        None

let reduce e = (* Fermeture de reduce1 *)
```

```
match reduce1 e with
    | None -> e
    | Some e' -> reduce e'
```

Remarque : Cet interprète n'est *pas très efficace*. Il passe son temps à *recalculer* le contexte puis à l'*"oublier"*. On peut faire mieux, avec un ***"zipper"***

Équivalence des deux sémantiques opérationnelles

Nous allons la montrer, ie $e \twoheadrightarrow v$ ssi $e \xrightarrow{\epsilon} v$

Lemme (passage au contexte des réductions) On suppose $e \to v$. Alors pour toute exp e_2 et valeur v :

1. $e\ e_2 \to e'\ e_2$
2. $v\ e \to v\ e'$
3. $let\ x = e\ in\ e_2 \to let\ x = e'\ in\ e_2$

Preuve : Pour *1.*: De $e \to e'$, on sait qu'il existe un contexte E tq $e = E(r)$, $e' = E(r')$ et $r \xrightarrow{\epsilon} r'$

On considère le contexte $E_1 = E\ e_2$. Alors

$$\frac{r \xrightarrow{\epsilon} r'}{E_1(r) \to E_1(r')} \text{ i.e. } \frac{r \xrightarrow{\epsilon} r'}{e\ e_2 \to e'\ e_2}$$

De même pour les autres cas. $\square$

Proposition "grands pas implique petits pas" : Si $e \twoheadrightarrow v$, alors $e \to^\star v$

Preuve : par récurrence sur la dérivation de $e \twoheadrightarrow v$.

Lemme : $v \twoheadrightarrow v$.

Preuve : trivial $\square$

Lemme : réduction et évaluation : Si $e \to e'$ et $e' \twoheadrightarrow v$, alors $e \twoheadrightarrow v$.

Preuve : On commence par les *réductions de tête* i.e. $e \xrightarrow{\epsilon} e'$ (donc contexte vide) Supposons par ex $e = (fun\ x \to e1)v_2$ et $e' = e_1[x \leftarrow v_2]$ On construit la dérivation en utilisant le lemme précédent et l'hypothèse $e' \twoheadrightarrow v$.

Montrons maintenant que si $e \xrightarrow{\epsilon} e'$ et $E(e') \twoheadrightarrow v$ alors $E(e) \twoheadrightarrow v$ (ie. réduction en profondeur) Par **récurrence structurelle sur** E (on vient de faire le cas $E = \square$)

Considérons, par exemple : $E = E'\ e_2$. On a $E(e') \twoheadrightarrow v$ ie. $E'(e')\ e_2 \twoheadrightarrow v$ Il suffit d'appliquer la forme de dérivation de $E'(e')\ e_2 \twoheadrightarrow v$. Ayant que, par **HR**, la propriété est vraie pour E', la forme est vraie. D'où $E'(e')\ e_2 \twoheadrightarrow v$, ie $E(e) \twoheadrightarrow v$

Les autres cas sont similaires. □

Proposition "petits pas implique grands pas" : Si $e \xrightarrow{\epsilon} v$, alors $e \twoheadrightarrow v$

Preuve : Récurrence finie sur les petits pas, par les lemmes précédents! □

ANNEXE]

On peut définir une **sémantique opérationnelle**, à *grands ou petits pas*, pour un langage avec des **traits impératifs**.

On associe typiquement un **état S** à l'espression évaluée/réduite

$S, e \twoheadrightarrow S', v$ ou bien $S, e \rightarrow S', e'$

Exemple :

$$\frac{S, e \twoheadrightarrow S', v}{S, x := e \twoheadrightarrow S' \oplus \{x \mapsto v\}, \mathrm{void}}$$

S peut être décomposé en une pile, un tas, etc.