Principes des lang. de progr.
INE 213

François Pessaux
(Majeure paternité : Michel Mauny)

ENSTA ParisTech, édition 2023-2024

`prénom.nom@ensta.fr`

Sémantique opérationnelle

Sémantique dénotationnelle *vs.* opérationnelle

Sémantique dénotationnelle

- valeurs abstraites
- le sens, pas le calcul
- quid des programmes qui ne terminent pas ?
- quid des programmes qui produisent une erreur d'exécution ?

Sémantique opérationnelle

- utilise des valeurs concrètes (syntaxiques)
- explicite les étapes du calcul
- procède par réduction (transformation syntaxique) du programme initial
- décrite par des catégories syntaxiques + des règles d'inférence
- G. Plotkin, 1981 ; G. Kahn 1985 (sémantique « naturelle »)

Noyau fonctionnel : évaluation stricte

Le langage

$c \in \mathrm{Const}$	*Constantes, incluant les fonctions primitives*
$x \in \mathrm{Id}$	*Identificateurs*
$e \in \mathrm{Exp}$	*Expressions*

Syntaxe

$$
\begin{array}{rcl}
e & ::= & c \mid x \mid e_1 + e_2 \mid e_1 = e_2 \\
 & \mid & \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \\
 & \mid & e_1 \; e_2 \\
 & \mid & \textbf{fun}\, x \;\to\; e \\
 & \mid & \textbf{let } x = e_1 \textbf{ in } e_2 \\
 & \mid & \textbf{let rec } f(x) = e_1 \textbf{ in } e_2
\end{array}
$$

Noyau fonctionnel, évaluation stricte

Valeurs

$$v \quad ::= \quad c \mid \mathsf{Valf}(x, e, \rho) \mid \mathsf{Valfr}(f, x, e, \rho)$$

- ρ environnement : $x \in Id \rightarrow v$
- $\rho \oplus [x \mapsto v]$: extension de ρ
  - $(\rho \oplus [x \mapsto v])(x) = v$
  - $(\rho \oplus [x \mapsto v])(y) = \rho(y)$ si $y \neq x$
- Valf et Valfr : valeurs fonctionnelles (fermetures ou *closures*)

Réponses

$$r \quad ::= \quad v \mid \mathsf{Erreur}$$

Règles d'évaluation

définissent des « jugements » de la forme

$$\rho \vdash e \Rightarrow r$$

« Dans l'env. ρ, l'évaluation de l'expression e produit la réponse r. »

Langage fonctionnel, strict : règles d'inférence

Constantes, variables, fonctions

$$\rho \vdash c \Rightarrow c \text{ (Const)} \qquad \frac{x \in \mathrm{dom}(\rho)}{\rho \vdash x \Rightarrow \rho(x)} \text{ (Ident)}$$

$$\rho \vdash (\textbf{fun}\, x \;\rightarrow\; e) \Rightarrow \mathrm{Valf}(x, e, \rho) \text{ (Fun)}$$

Conditionnelle

$$\frac{\rho \vdash e_1 \Rightarrow T \qquad \rho \vdash e_2 \Rightarrow r}{\rho \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \Rightarrow r} \text{ (IfTrue)}$$

$$\frac{\rho \vdash e_1 \Rightarrow F \qquad \rho \vdash e_3 \Rightarrow r}{\rho \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \Rightarrow r} \text{ (IfFalse)}$$

Langage fonctionnel, strict : règles

Primitives

$$\frac{\rho \vdash e_1 \Rightarrow n_1 \qquad \rho \vdash e_2 \Rightarrow n_2 \qquad n = n_1 + n_2}{\rho \vdash (e_1 + e_2) \Rightarrow n} \text{ (Plus)}$$

$$\frac{\rho \vdash e_1 \Rightarrow v_1 \qquad \rho \vdash e_2 \Rightarrow v_2 \qquad v_1 = v_2}{\rho \vdash (e_1 = e_2) \Rightarrow T} \text{ (EgTrue)}$$

$$\frac{\rho \vdash e_1 \Rightarrow v_1 \qquad \rho \vdash e_2 \Rightarrow v_2 \qquad v_1 \neq v_2}{\rho \vdash (e_1 = e_2) \Rightarrow F} \text{ (EgFalse)}$$

Langage fonctionnel, strict : règles

Déclarations

$$\frac{\rho \vdash e_1 \Rightarrow v_1 \qquad \rho \oplus [x \mapsto v_1] \vdash e_2 \Rightarrow r}{\rho \vdash (\textbf{let } x = e_1 \textbf{ in } e_2) \Rightarrow r} \text{ (Let)}$$

$$\frac{\rho \oplus [f \mapsto \mathrm{Valfr}(f, x, e_1, \rho)] \vdash e_2 \Rightarrow r}{\rho \vdash (\textbf{let rec } f(x) = e_1 \textbf{ in } e_2) \Rightarrow r} \text{ (Letrec)}$$

Langage fonctionnel, strict : règles

Applications

$$\frac{\begin{array}{c}\rho \vdash e_1 \Rightarrow \mathrm{Valf}(x, e_0, \rho_0) \\ \rho \vdash e_2 \Rightarrow v_2 \qquad \rho_0 \oplus [x \mapsto v_2] \vdash e_0 \Rightarrow r\end{array}}{\rho \vdash (e_1 e_2) \Rightarrow r}\ (\mathrm{AppF})$$

$$\frac{\begin{array}{c}\rho \vdash e_1 \Rightarrow \mathrm{Valfr}(f, x, e_0, \rho_0) \qquad \rho \vdash e_2 \Rightarrow v_2 \\ \rho_0 \oplus [f \mapsto \mathrm{Valfr}(f, x, e_0, \rho_0)] \oplus [x \mapsto v_2] \vdash e_0 \Rightarrow r\end{array}}{\rho \vdash (e_1 e_2) \Rightarrow r}\ (\mathrm{AppFR})$$

Les règles auxquelles vous avez échappé

Production d'erreurs et propagation

$$\frac{x \notin \text{dom}(\rho)}{\rho \vdash x \Rightarrow \text{Erreur}}\ (\text{IdentErr}) \qquad \frac{\rho \vdash e_1 \Rightarrow r \notin \{T, F\}}{\rho \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \Rightarrow \text{Erreur}}\ (\text{IfErr})$$

$$\frac{\rho \vdash e_1 \Rightarrow r \notin \mathbb{N}}{\rho \vdash (e_1 + e_2) \Rightarrow \text{Erreur}}\ (\text{PlusErrL}) \qquad \frac{\rho \vdash e_1 \Rightarrow n_1 \qquad \rho \vdash e_2 \Rightarrow r \notin \mathbb{N}}{\rho \vdash (e_1 + e_2) \Rightarrow \text{Erreur}}\ (\text{PlusErrR})$$

$$\frac{\rho \vdash e_1 \Rightarrow \text{Erreur}}{\rho \vdash (e_1 = e_2) \Rightarrow \text{Erreur}}\ (\text{EgErrL}) \qquad \frac{\rho \vdash e_1 \Rightarrow v_1 \qquad \rho \vdash e_2 \Rightarrow \text{Erreur}}{\rho \vdash (e_1 = e_2) \Rightarrow F}\ (\text{EgErrR})$$

$$\frac{\rho \vdash e_1 \Rightarrow \text{Erreur}}{\rho \vdash (\textbf{let } x = e_1 \textbf{ in } e_2) \Rightarrow \text{Erreur}}\ (\text{LetErr})$$

$$\frac{\rho \vdash e_1 \Rightarrow r \neq \text{Valf}(\_, \_, \_) \wedge r \neq \text{Valfr}(\_, \_, \_, \_)}{\rho \vdash (e_1\, e_2) \Rightarrow \text{Erreur}}\ (\text{AppErrL})$$

$$\frac{\rho \vdash e_1 \Rightarrow \text{Valf}(x, e_0, \rho_0) \qquad \rho \vdash e_2 \Rightarrow \text{Erreur}}{\rho \vdash (e_1\, e_2) \Rightarrow \text{Erreur}}\ (\text{AppFErrR})$$

$$\frac{\rho \vdash e_1 \Rightarrow \text{Valfr}(f, x, e_0, \rho_0) \qquad \rho \vdash e_2 \Rightarrow \text{Erreur}}{\rho \vdash (e_1\, e_2) \Rightarrow \text{Erreur}}\ (\text{AppFRErrR})$$

Arbres d'évaluation – dérivations

Arbres : axiomes = feuilles, règles d'inférence = nœuds

Posons $\overline{f} = \mathrm{Valf}(x, x, \emptyset)$ et $\rho_f = \emptyset \oplus [f \mapsto \overline{f}] = [f \mapsto \overline{f}]$.

$$\dfrac{\dfrac{f \in \mathrm{dom}(\rho_f)}{\rho_f \vdash f \Rightarrow \overline{f}} \qquad \rho_f \vdash 1 \Rightarrow 1 \qquad \dfrac{x \in \mathrm{dom}(\emptyset \oplus [x \mapsto 1])}{\emptyset \oplus [x \mapsto 1] \vdash x \Rightarrow 1}}{\rho_f \vdash f(1) \Rightarrow 1} \; (\mathrm{App})$$

$$\dfrac{(\mathrm{Fun}) \; \dfrac{}{\emptyset \vdash (\mathbf{fun}\, x \; \rightarrow \; x) \Rightarrow \overline{f}}}{\emptyset \vdash (\mathbf{let} \; f = \mathbf{fun}\, x \; \rightarrow \; x \; \mathbf{in} \; f(1)) \Rightarrow 1} \; (\mathrm{Let})$$

Données structurées

Couples et projections

$$e \quad ::= \quad \ldots \mid (e_1, e_2) \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e$$

Valeurs

$$v \quad ::= \quad \ldots \mid (v_1, v_2)$$

Règles

$$\frac{\rho \vdash e_1 \Rightarrow v_1 \qquad \rho \vdash e_2 \Rightarrow v_2}{\rho \vdash (e_1, e_2) \Rightarrow (v_1, v_2)}$$

$$\frac{\rho \vdash e \Rightarrow (v_1, v_2)}{\rho \vdash \mathsf{fst}\ e \Rightarrow v_1} \qquad \frac{\rho \vdash e \Rightarrow (v_1, v_2)}{\rho \vdash \mathsf{snd}\ e \Rightarrow v_2}$$

De l'évaluation (formelle) à l'interprétation (mécanique)

Implémenter les valeurs de la sémantique opérationnelle :

$$v \quad ::= \quad c \mid \mathsf{Valf}(x, e, \rho) \mid \mathsf{Valfr}(f, x, e, \rho)$$

par :

```
type semopval =
  | Intval of int
  | Boolval of bool
  | Stringval of string
  | ...
  | Funval of { param: string; body: expr; env: environment }
  | Funrecval of
     { fname: string; param: string; body: expr; env: environment }

and environment = (string * semopval) list ;;
```

On considère qu'une *réponse* sera ou bien une valeur, ou alors la levée d'une exception OCaml.

De l'évaluation à l'interprétation

Puis « lire » les axiomes et règles d'inférence comme des cas d'une fonction d'évaluation :

```
val eval : Pcfast.expr → environment → semopval

let rec eval t rho = match t with ...

  | Pcfast.EInt n → Intval n                               (* ρ ⊢ c ⇒ c *)
  | Pcfast.EString s → Stringval s                         (* ρ ⊢ c ⇒ c *)

  | Pcfast.EIdent x → begin                                   x ∈ dom(ρ)
      try List.assoc x rho with                          (* ────────────── *)
          Not_found →                                       ρ ⊢ x ⇒ ρ(x)
            error (Printf.sprintf "Unbound variable %s." x)
    end
  | ...
```

De l'évaluation à l'interprétation

Les axiomes et règles d'inférence comme des cas d'une fonction d'évaluation :

```
let rec eval t rho = match t with ...

| Pcfast.EFun (x, e) →                    (* ρ ⊢ (fun x → e) ⇒ Valf(x, e, ρ) *)
        Funval { param = x; body = e; env = rho }
| Pcfast.EIf (e1, e2, e3) → begin
    match eval e1 rho with
      | Boolval true → eval e2 rho
      | Boolval false → eval e3 rho
      | _ → error "Bool value expected"
  end
| ...
```

$$(*\ \frac{\rho \vdash e_1 \Rightarrow T \qquad \rho \vdash e_2 \Rightarrow r}{\rho \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \Rightarrow r}\ *)$$

$$(*\ \frac{\rho \vdash e_1 \Rightarrow F \qquad \rho \vdash e_3 \Rightarrow r}{\rho \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \Rightarrow r}\ *)$$

De l'évaluation à l'interprétation

Les axiomes et règles d'inférence comme des cas d'une fonction d'évaluation :

```
let rec eval t rho = match t with ...

| Pcfast.EBinop ("+", e1, e2) → begin
    match (eval e1 rho, eval e2 rho) with
    | (Intval n1, Intval n2) → Intval (n1 + n2)
    | _ → error "Integer values expected"
  end

| ...
```

$$(*\ \frac{\rho \vdash e_1 \Rightarrow n_1 \qquad \rho \vdash e_2 \Rightarrow n_2}{\rho \vdash (e_1 + e_2) \Rightarrow n_1 + n_2}\ *)$$

De l'évaluation à l'interprétation

Les axiomes et règles d'inférence comme des cas d'une fonction d'évaluation :

```
let rec eval t rho = match t with ...

| Pcfast.ELet (x, e1, e2) →
    let v1 = eval e1 rho in
    eval e2 ((x,v1)::rho)
```

$$(*\ \frac{\rho \vdash e_1 \Rightarrow v_1 \qquad \rho \oplus [x \mapsto v_1] \vdash e_2 \Rightarrow r}{\rho \vdash (\textbf{let } x = e_1 \textbf{ in } e_2) \Rightarrow r}\ *)$$

```
| Pcfast.ELetrec (f, x, e1, e2) →
    let vf =
      Funrecval { fname = f; param = x; body = e1; env = rho } in
    eval e2 ((f, vf) :: rho)
```

$$(*\ \frac{\rho \oplus [f \mapsto \text{Valfr}(f, x, e_1, \rho)] \vdash e_2 \Rightarrow r}{\rho \vdash (\textbf{let rec } f(x) = e_1 \textbf{ in } e_2) \Rightarrow r}\ *)$$

```
| ...
```

De l'évaluation à l'interprétation

Des axiomes et règles d'inférence aux cas d'une fonction d'évaluation

```
let rec eval t rho = match t with
  ...
| Pcfast.EApp (e1, e2) ->
   let (vf, v2) = (eval e1 rho, eval e2 rho) in begin
   match vf with
```

$$(*\ \frac{\rho \vdash e_1 \Rightarrow \mathsf{Valf}(x, e_0, \rho_0) \qquad \rho \vdash e_2 \Rightarrow v_2 \qquad \rho_0 \oplus [x \mapsto v_2] \vdash e_0 \Rightarrow r}{\rho \vdash (e_1\, e_2) \Rightarrow r}\ *)$$

```
    | Funval { param = x; body = e0; env = rho0 } ->
       eval e0 ((x, v2) :: rho0)
```

$$(*\ \frac{\rho \vdash e_1 \Rightarrow \mathsf{Valfr}(f, x, e_0, \rho_0) \qquad \rho \vdash e_2 \Rightarrow v_2 \qquad \rho_0 \oplus [f \mapsto \mathsf{Valfr}(f, x, e_0, \rho_0)] \oplus [x \mapsto v_2] \vdash e_0 \Rightarrow r}{\rho \vdash (e_1\, e_2) \Rightarrow r}\ *)$$

```
    | Funrecval { fname = f; param = x; body = e0; env=rho0 } ->
       eval e0 ((x, v2) :: (f, vf) :: rho0)
    | _ -> error "Expecting a functional value"
  end ;;
```

De l'interprétation à la compilation

Interprète

- + environnements, valeurs
- − gestion du contrôle par le langage hôte
  - $e_1 + e_2 \rightsquigarrow (\texttt{eval}\ e_1) + (\texttt{eval}\ e_2)$: ordre `eval`'s non spécifié

(Un modèle de) Compilateur

- + environnements, valeurs
- + gestion explicite du contrôle (pile)
  - $e_1 + e_2 \rightsquigarrow r_1 \leftarrow (\texttt{eval}\ e_2); r_2 \leftarrow (\texttt{eval}\ e_1); r_3 \leftarrow \texttt{add}\ r_1, r_2$

Machine abstraite

- code, pile, registre, (mémoire, *etc.*)
- instruction : état → état

Compilation

Le langage

$$\begin{array}{rcl} e & ::= & c \mid x \mid e_1 e_2 \mid \textbf{fun}\, x \to e \mid \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \\ & \mid & e_1 + e_2 \mid e_1 = e_2 \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \end{array}$$

Codage de *de Bruijn* des occurrences de variables

- occurrence de `x` dans un contexte où *n* lieurs («**fun** $\mathtt{y} \to$», «**let** $\mathtt{y}\ =$» ou «**let** $\mathtt{f(y)}\ =$») séparent cette occurrence de son lieur
- correspond à un accès à une profondeur *n* dans un environnement : `Access(n)`

Exemples

$$\begin{array}{ll} \textbf{fun}\ \mathtt{x} \to & \mathtt{x} \\ & \mathtt{Access(0)} \end{array}$$

$$\begin{array}{lr} \textbf{fun}\ \mathtt{x} \to & \textbf{let}\ \mathtt{y} = \ldots\ \textbf{in}\ \textbf{let}\ \mathtt{z} = \ldots\ \textbf{in}\ \textbf{fun}\ \mathtt{t} \to\ \ \ldots\ \mathtt{x}\ \ldots \\ & \mathtt{Access(3)} \end{array}$$

Compilation

État de la machine (registre, code, pile)

- registre : contient une valeur ou un environnement
- code : instructions
- pile : chaque élément est valeur, ou env., ou adresse de retour

État initial / début de fonction : registre contient l'environnement.

Les instructions

i ::=	Loadi (n) | Loadb (b)	chargement de cst dans le registre
|	Plus | Equal	opérations binaires (arith, tests, ...)
|	Access (n)	accès dans env.
|	Branch (c_1, c_2)	branchement
|	Push | Swap	empile, échange
|	Mkclos (c)	constr. de valeur fonctionnelle
|	Apply	application

La machine dans tous ses états

État		État suivant
$(r, \mathtt{Loadi}\ (n) :: c, p)$	$\Rightarrow$	(n, c, p)
$(r, \mathtt{Loadb}\ (b) :: c, p)$	$\Rightarrow$	(b, c, p)
$(n, \mathtt{Plus} :: c, m :: p)$	$\Rightarrow$	$(\overline{n+m}, c, p)$
$(n, \mathtt{Equal} :: c, m :: p)$	$\Rightarrow$	$(\overline{n=m}, c, p)$
$(\mathrm{true}, \mathtt{Branch}\ (c_1, c_2) :: c, r :: p)$	$\Rightarrow$	$(r, c_1, c :: p)$
$(\mathrm{false}, \mathtt{Branch}\ (c_1, c_2) :: c, r :: p)$	$\Rightarrow$	$(r, c_2, c :: p)$
$(r, \mathtt{Push} :: c, p)$	$\Rightarrow$	$(r, c, r :: p)$
$(r_1, \mathtt{Swap} :: c, r_2 :: p)$	$\Rightarrow$	$(r_2, c, r_1 :: p)$
$(r, \mathtt{Mkclos}\ (c_1) :: c, p)$	$\Rightarrow$	$(\langle c_1, r \rangle, c, p)$
$(v, \mathtt{Apply} :: c, \langle c_0, r_0 \rangle :: p)$	$\Rightarrow$	$((v \oplus r_0), c_0, c :: p)$
$(\rho, \mathtt{Access}\ (n) :: c, p)$	$\Rightarrow$	$(\overline{\rho(n)}, c, p)$
$(r, [\ ], c :: p)$	$\Rightarrow$	(r, c, p)

La machine dans tous ses états (en OCaml)

```
let next state = match state with
| (r,         Loadi n::c,         p)           → (I(n), c, p)
| (r,         Loadb b::c,         p)           → (B(b), c, p)
| (I(n),      Plus::c,            I(m)::p)     → (I(n+m), c, p)
| (I(n),      Equal::c,           I(m)::p)     → (B(n=m), c, p)
| (B(true),   Branch(c1,c2)::c,   r::p)        → (r, c1, A(c)::p)
| (B(false),  Branch(c1,c2)::c,   r::p)        → (r, c2, A(c)::p)
| (r,         Push::c,            p)           → (r, c, r::p)
| (r1,        Swap::c,            r2::p)       → (r2, c, r1::p)
| (r,         Mkclos(c1)::c,      p)           → (C(c1, r), c, p)
| (v,         Apply::c,           C(c0,r0)::p) → (E(v,r0), c0, A(c)::p)
| (E(v::_),    Access(0)::c,       p)           → (v, c, p)
| (E(v::q),    Access(n)::c,       p)           → (q, Access(n−1)::c, p)
| (r,         [ (* ret *) ],      A(c)::p)     → (r, c, p)
| (r,         [],                 [])         → raise (Success r)
| _ → error "Error: invalid state (machine stopped)"
```

La compilation

Compilation du noyau

$$
\begin{array}{lcl}
\llbracket n \rrbracket_\rho & = & \mathtt{Loadi}\ (n) \\
\llbracket b \rrbracket_\rho & = & \mathtt{Loadb}\ (b) \\
\llbracket x \rrbracket_\rho & = & \mathtt{Access}\ (n) \\
 & & \text{où } n \text{ est la profondeur de } x \text{ dans } \rho \\
\llbracket e_1 e_2 \rrbracket_\rho & = & \mathtt{Push};\llbracket e_1 \rrbracket_\rho;\mathtt{Swap};\llbracket e_2 \rrbracket_\rho;\mathtt{Apply} \\
\multicolumn{3}{l}{\llbracket \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \rrbracket_\rho} \\
 & = & \mathtt{Push};\llbracket e_1 \rrbracket_\rho;\mathtt{Branch}\ (\llbracket e_2 \rrbracket_\rho, \llbracket e_3 \rrbracket_\rho) \\
\llbracket \mathbf{fun}\, x\ \rightarrow\ e \rrbracket_\rho & = & \mathtt{Mkclos}\ (\llbracket e \rrbracket_{x,\rho}) \\
\llbracket \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \rrbracket_\rho & = & \llbracket (\mathbf{fun}\, x\ \rightarrow\ e_2) e_1 \rrbracket_\rho
\end{array}
$$

Compilation des définitions récursives ?

- Laissée en exercice (pas si simple)

Conclusion

La sémantique opérationnelle

- précise comment s'effectuent les calculs
- se rapproche des interprètes

La compilation

- consiste en premier lieu à expliciter le contrôle

⇒ manipulation de pile

- Nombreux autres traitements non abordés :
  - allocation de registres,
  - optimisations,
  - sélection d'instructions (production de code assembleur),
  - ...