ENVIRONMENTAL MODEL

Abbiamo visto interpreti basati su substitution model

⇝ meccanismo di calcolo è la sostituzione

$$\frac{e' \Rightarrow v' \qquad e[v'/x] \Rightarrow v}{(\text{fun } x \to e)\ e' \Rightarrow v}$$

Modello per sostituzione è però

- modello matematico (concettualmente utile)
- difficile da implementare (⇝ capture avoidance)
- inefficiente $e[v/x]$ ⇝ sostituzione simultanea

<u>sostituzione</u> ⇝ buon modello simbolico, ma non realistico modello della macchina

- separazione programma e dati
  - → area memoria codice
  - → area memoria dati

- non viene eseguita sostituzione a runtime → dati lasciati in memoria e presi solo quando necessario

```
let x = 42 in
  let y = 10 in
    x + y
```

scope x

scope y

x	42

x	42
y	10

→ a runtime viene dedicata una zona di memoria (e.g. registro) in cui viene salvata (nome, valore)

→ a questo punto si guarda qual è il valore di x e di y

Def. Ambiente dinamico = $\underbrace{\text{zone di memoria}}_{\text{astrazione}}$ create a run-time

"ambiente statico contiene: tipi"

durante la dichiarazione di un nome e che contengono il valore dato al nome dalla dichiarazione nello scope della dichiarazione

Modello semantico

$\langle \eta, e \rangle \Downarrow v$

ambiente

$\underbrace{\eta : \text{ident} \to \text{val option}}_{\text{funzione parziale}}$

Siamo passati da riduzione simbolica a dinamica di configurazioni macchina

$\langle \eta, e \rangle \to \langle \eta', e' \rangle$

NB. Non c'è memoria!

ENVIRONMENTAL - MODEL BIG-STEP SEMANTICS

$$\langle \eta, e \rangle \Downarrow v$$

Substitution - model BIG - STEP SEMANTICS

$$e \Rightarrow v$$

<u>Teorema</u> (Correttezza). $\langle \eta, e \rangle \Downarrow v$ sse $e[v_1 \dots v_n / x_1 \dots x_n] \Rightarrow v$

dove $\text{var\_libere}(e) = \{x_1 \dots x_n\}$

$\eta(x_i) = \text{Some } v_i$

Esempio di regole semantiche

$$\langle \eta, x \rangle \Downarrow \eta(x)$$

operazione di: <u>binding</u>: estendiamo/aggiorniamo l'ambiente assegnando un valore ad x

$$\frac{\langle \eta, e_1 \rangle \Downarrow v_1 \qquad \langle \eta[x \mapsto v_1], e_2 \rangle \Downarrow v_2}{\langle \eta, \text{let } x = e_1 \text{ in } e_2 \rangle \Downarrow v_2}$$

Qui non c'è $\eta[x \mapsto v_1]$ perché lo <u>scope</u> di x in $\text{let } x = e_1 \text{ in } e_2$ è <u>solamente</u> e_2.

$$\frac{\langle \eta, e_1 \rangle \Downarrow v_1 \qquad \langle \eta, e_2 \rangle \Downarrow v_2 \qquad \text{primitive\_op}(+, v_1, v_2) = v}{\langle \eta, e_1 + e_2 \rangle \Downarrow v}$$

IMPLEMENTAZIONE INTERPRETE SIMPL

② Implementazione ambiente

$$\text{type } \underbrace{\alpha \text{ env}}_{\text{polimorfo}} = \text{ident} \rightarrow \alpha \text{ option}$$

Intuizione $\quad \eta\ x = \begin{cases} \text{Some } v & \text{se } x \text{ è definito in } \eta \text{ e ha valore } v \\ \text{None} & \text{altrimenti} \end{cases}$

Per accedere al valore di una variabile in un ambiente η

$$\eta\ x$$

ha tipo α env (= ident → a option)

Per aggiungere / aggiornare

```
let bind_env η x v = fun y -> if (x=y) then v
                                       else η y
```

② Implementazione sintassi

```
type ident = ...
type exp = ...
type val = ...
```
} valori esprimibili (risultato valutazione espressioni)

③ Implementazione semantica

3.1 Operazioni primitive

```
let primitive_op : binop -> val -> val -> val =
    fun op v1 v2 -> match op, v1, v2 with
        | Add, IntVal n1, IntVal n2 ->
                IntVal (n1+n2)
        ⋮
```

3.2. Eval

```
let eval (e₀ : exp) : val =
   let eval_env (η : val env)(e : exp) : val =
      match e with
        | Var x  ->  match (η x) with
                       | None  -> failwith ".. "
                       | Some v  ->  v

        | Int n  ->  IntVal n
          :                                          ambiente ?
        | Binop bop v₁ v₂  ->  primitive-op bop v₁ v₂
        | Let (x, e₁, e₂)  ->  eval_rec (bind_env η x (eval_rec η e₁)) e₂

   in eval_env (fun x -> None) e
```

INTERPRETE PER λ-SimPL

Aggiungiamo a SimPL il λ-calcolo ($\leadsto$ quasi miniCaml)

$$
\begin{aligned}
e ::= \ & x \\
\mid \ & n \\
\mid \ & b \\
\mid \ & \text{bop } e \ e \\
\mid \ & \text{if } e \text{ then } e \text{ else } e \\
\mid \ & \text{let } x = e \text{ in } e \\
\mid \ & \text{lam } x \to e \qquad\qquad (\text{fun } x \to e) \\
\mid \ & \text{app } e \ e \qquad\qquad (e \ e)
\end{aligned}
$$

$$
\begin{aligned}
v ::= \ & x \\
\mid \ & n \\
\mid \ & b \\
\mid \ & \text{lam } x \to e
\end{aligned}
$$

$$\text{bop} ::= + \mid * \mid \dots$$

Dobbiamo dare semantica big-step basata su ambiente

Tentativo 1

$$\langle \eta, \mathrm{Lam}(x,e)\rangle \Downarrow \mathrm{Lam}(x,e)$$

(funzioni sono valori)

$\mathrm{Lam}(x,e)\ w \rightsquigarrow e[w/x]$

$$\frac{\langle \eta, e_1\rangle \Downarrow \mathrm{Lam}(x,e) \qquad \langle \eta, e_2\rangle \Downarrow w \qquad \overbrace{\langle \eta[x \mapsto w], e\rangle \Downarrow v}}{\langle \eta, \mathrm{App}(e_1,e_2)\rangle \Downarrow v}$$

Cosa succede se eseguiamo

```
let x = 1 in
let f = lam y -> x
let x = 2 in
f 0
```

scope x

scope x

→ valuta a 1 o a 2 ?

$\text{let } x = 1 \text{ in}$

$\{x \mapsto 1\}$

$\text{let } f = \text{lam } y \to x \text{ in}$

$\{x \mapsto 1;\ f \mapsto (\text{lam } y \to x)\}$

$\text{let } x = 2 \text{ in}$

$\{x \mapsto 2;\ f \mapsto (\text{lam } y \to x)\}$

$f\,0$

$$\langle x \mapsto 2;\ f \mapsto \text{lam } y \to x,\ f\,0 \rangle \Downarrow ?$$

$$\dfrac{\dfrac{\dfrac{}{\langle x \mapsto 1,\ \text{lam } y \to x \rangle \Downarrow \text{lam } y \to x} \qquad \dfrac{\langle x \mapsto 1;\ f \mapsto \text{lam } y \to x,\ 2 \rangle \Downarrow 2}{\langle x \mapsto 1;\ f \mapsto \text{lam } y \to x,\ \text{let } x = 2 \text{ in } f\,0 \rangle \Downarrow ?}}{\langle x \mapsto 1,\ \text{let } f = \text{lam } y \to x \text{ in } (\text{let } x = 2 \text{ in } f\,0) \rangle \Downarrow ?} \qquad \langle \cdot, 1 \rangle \Downarrow 1}{\langle \cdot, e \rangle \Downarrow ?}$$

La nostra regola per le funzioni implementa lo scope dinamico

Regola dello scope dinamico

Il corpo di una funzione viene valutato nell'ambiente presente al momento della chiamata della funzione, non nell'ambiente presente al momento della definizione della funzione

Regola dello scope statico (o lessicale)

OCaml usa scope lessicale (come quasi tutti i linguaggi)

→ Come implementare questa regola?
Ogni volta che definiamo una funzione, dobbiamo salvare l'ambiente;
Ogni volta che chiamiamo una funzione, dobbiamo recuperare l'ambiente presente al momento della sua definizione

Perchè scope lessicale?

- Facile da "prevedere"
- Rispetta buoni principi semantici

$$\left(\begin{array}{l} \text{let } x = 1 \text{ in} \\ \text{let } f = \text{lam } y \to x \quad \text{in} \\ \text{let } x = 2 \\ \quad \text{in } f() \end{array}\right)$$

cosa succede se rinomino variabile legata x?

$$\left(\begin{array}{l} \text{let } x = 1 \text{ in} \\ \text{let } f = \text{lam } y \to x \quad \text{in} \\ \text{let } z = 2 \\ \quad \text{in } f() \end{array}\right)$$

⇝ semantica dipendente dalla scelta delle variabili legate

Implementazione Scope Lessicale

Idea principale: il valore associato a una funzione non è una funzione, ma una struttura detta chiusura (closure) che contiene la funzione e l'ambiente in cui essa è definita

```
type val =
  | IntVal of int
  | BoolVal of bool
  | Closure of ident * exp * val env        (ricorsivo)
```

NB. Chiusure **non** sono valori denotabili (ma esprimibili)

Semantica operazionale

$$\langle \eta, Lam(x, e) \rangle \Downarrow Closure(x, e, \eta)$$

$$\frac{\langle \eta, e_1 \rangle \Downarrow Closure(x, e, \delta) \qquad \langle \eta, e_2 \rangle \Downarrow w \qquad \langle \delta[x \mapsto w], e \rangle \Downarrow v}{\langle \eta, App(e_1, e_2) \rangle \Downarrow v}$$

```
let eval (e₀ : exp) : val =
  let eval_env (η : val env) (e : exp) : val =
    match e with
      | Var x -> match (η x) with
                  | None -> failwith ".. "
                  | Some v -> v
      | Int n -> IntVal n
        :
      | Binop bop v₁ v₂ -> primitive-op bop v₁ v₂
      | Let (x, e₁, e₂) -> eval_rec (bind_env η x (eval_rec η e₁)) e₂
      | Lam (x, e) -> Closure (x, e, η)
      | App (e₁, e₂) -> match (eval_env η e₁) with
          | Closure (x, f, δ) -> eval_env (bind δ x (eval_env η e₂)) f
          | _ -> failwith ...
  in eval_env (fun x -> None) e
```

Funzioni Ricorsive

$$e ::= \dots$$
$$\quad \mid \texttt{let\_rec}\ f\ x = e\ \texttt{in}\ e$$

```
type exp =
        :
      | LetRec of ident * ident * exp * exp

type val =
       :
     | ClosureRec of ident * ident * exp * env
```

Trattiamo le funzioni ricorsive come funzioni, ma teniamo traccia del loro nome per poter mettere nell'ambiente la chiamata ricorsiva

Semantica

$$\frac{\langle \eta[f \mapsto \text{ClosureRec}(f, x, e, \eta)],\ e' \rangle \Downarrow v}{\langle \eta,\ \text{LetRec}(f, x, e, e') \rangle \Downarrow v}$$

$$\frac{\langle \eta, e_1 \rangle \Downarrow \overbrace{\text{ClosureRec}(f, x, e, \delta)}^{clo} \qquad \langle \eta, e_2 \rangle \Downarrow v \qquad \langle \delta[f \mapsto clo,\ x \mapsto v], e \rangle \Downarrow w}{\langle \eta,\ \text{App}(e_1, e_2) \rangle \Downarrow w}$$

Interprete

⋮

| LetRec $(f, x, e_1, e_2) \rightarrow$ eval_env (bind η f ClosureRec(f, x, e_1, η)) e_2

⋮

| App $(e_1, e_2) \rightarrow$ match (eval_env η e_1) with

| Closure $(x, e', \delta) \rightarrow$ eval_env (bind δ x (eval_env η e_2)) e'

| ClosureRec $(f, x, e', \delta) \rightarrow$

let δ' = bind δ f ClosureRec (f, x, e', δ)

in eval_env (bind δ' x (eval_env η e_2)) e'

D: fatto questa è la parte diversa da Closure

⋮