Ci focalizziamo ora sull'interprete di un linguaggio funzionale "core"

Prima vedremo un frammento base a cui poi aggiungiamo funzioni

Linguaggio SimPL

e ::= x | n | b | e binop e (n ∈ ℕ, b ∈ {true, false})
| if e Then e else e
| let x = e in e

binop ::= + | * | ≤

v ::= n | b

Visto che il linguaggio consiste solamente di espressioni ed è puro, possiamo vedere l'interprete come la funzione eval

```
type binop =
  | Plus
  | Times
  | Leq

type exp =
  | Var of identifier
  | Int of int
  | Bool of bool
  | Binop of binop * exp * exp
  | If of exp * exp * exp
  | Let of identifier * exp * exp

type identifier = string
```

```
eval (Int 5) =
    IntVal 5
```

↳ anche `exp * exp * exp` è ok

```
type val =
  | IntVal of int
  | BoolVal of bool
```

in alternativa

```
let is_value (e:exp) : bool =
    match e with
      | Int  -> true
      | Bool -> true
      | _    -> false
```

Quindi vogliamo

$eval : exp \to val$

(oppure $eval : exp \to exp$ tale che $is\_value\ (eval\ e) == true$)

Come possiamo scrivere, in generale, $eval$?

→ semantica operazionale

Un interprete è una implementazione della semantica operazionale ⇝ semantica dinamica

Fino ad ora, abbiamo visto 2 st.li di sem. op.

Small-step $e \to e'$

passo di computazione singolo

$$\frac{e_1 \to e_1'}{\texttt{let } x = e_1 \texttt{ in } e_2 \to \texttt{let } x = e_1' \texttt{ in } e_2}$$

$$\frac{}{\texttt{let } x = v \texttt{ in } e \to e[v/x]}$$

Big-step $e \Rightarrow v$ (oppure $e \Downarrow v$)

comportamento globale

$$\frac{e_1 \Rightarrow v \qquad e_2[v/x] \Rightarrow w}{\texttt{let } x = e_1 \texttt{ in } e_2 \Rightarrow w}$$

Correttezza

$e \to^* v$

sse

$e \Rightarrow v$

$e \in Values$

sse

$e \not\to$

$$\frac{\text{Premessa}_1 \quad \dots \quad \text{Premessa}_n}{\text{Conclusione}}$$

Si ha Conclusione se si ha

Premessa 1
e
Premessa 2
e
⋮
e
Premessa n

Premessa 1 & ... & Premessa n
$\Longrightarrow$ Conclusione

NB. Solitamente la semantica big-step è non deterministica

$$\frac{e_1 \Rightarrow \text{Int } m_1 \qquad e_2 \Rightarrow \text{Int } m_2}{e_1 \text{ ADD } e_2 \Rightarrow \text{Int } (m_1 + m_2)}$$

Quale eseguire prima?

```
eval (ADD (e1, e2)) =
    let v1 = eval e1   (1)
        v2 = eval e2   (2)
    in
      primitive ADD v1 v2
```

vs

```
eval (ADD (e1, e2)) =
    let v2 = eval e1   (2)
        v1 = eval e2   (1)
    in
      primitive ADD v1 v2
```

Interprete = determinizzazione $\Rightarrow$

GESTIONE VARIABILI

let x = 5 in x + x → 5 + 5

Cosa succede se una espressione contiene variabili non legate ?

eval x
↳ la variabile è unbound; non possiamo fare nulla

Def Un programma è una espressione non contenente variabili libere

eval : exp → val
↳ solleva eccezioni;

(oppure
eval : exp → val option)

eval x =
fail with "variabile non legata"

Un programma può contenere variabili legate, che sono introdotte tramite espressioni let-in (↝ cf. dichiarazioni)

Possibili gestioni: variabili legate

① Sostituzione

Ogni volta che dichiariamo una variabile x con valore v, sostituiamo **tutte** le occorrenze libere di x con v nel rimanente corpo del programma

$$\texttt{let } x = v \texttt{ in } e \longrightarrow e[v/x]$$

matematicamente comodo ma poco efficiente

② Ambiente

Un ambiente tiene traccia dei valori assegnati alle variabili

$$(\eta, \texttt{let } x = v \texttt{ in } e) \longrightarrow (\eta[x \mapsto v], e)$$

$$(\eta, x) \longrightarrow (\eta, \eta(x))$$

$$(\eta, \texttt{let } x = 5 \texttt{ in } x + x) \longrightarrow (\eta[x \mapsto 5], x + x) \longrightarrow (\eta[x \mapsto 5], 5 + x)$$

Interprete basato su sostituzione

```
let rec eval (e : exp) : val = match e with
    | Int n -> ValInt n
    | Bool b -> ValBool b
    | Var _ -> failwith "unbound variable"
    | Binop (bop, e1, e2) ->
            let v1 = eval e1
                v2 = eval e1
            in primitive_op bop v1, v2
    | If (e1, e2, e3) -> match (eval e1) with
            | ValBool true -> eval e2
            | ValBool false -> eval e3
            | _ -> failwith "type error"
    | Let (x, e1, e2) -> let v = eval e1
                         in eval (subst e2 v x)
```

} operazione primitiva

} error. "static:"

} sostituzione come operazione primitiva

```
let primitive_op (bop : binop) (v1 : val) (v2 : val) : val =
  match bop, v1, v2 with
    | Add, IntVal a, IntVal b -> IntVal (a+b)
    | Mult, IntVal a, IntVal b -> IntVal (a*b)
    | Leq, IntVal a, IntVal b -> BoolVal (a ≤ b)
    | _ -> failwith "..."
```

chiamata per valore

```
let subst (e : exp) (v : val) (x : identifier) : exp =
  match e with
    | Var y -> if x = y then v else y
    | Int n -> Int n
    | Bool b -> Bool b
    | Binop (bop, e1, e2) -> Binop (bop, subst e1 v x, subst e2 v x)
    | If (e1, e2, e3) -> If (subst e1 v x, subst e2 v x, subst e3 v x)
    | Let (x, e1, e2) -> ???
```

e' = e[v/x]

(x ADD y)[5/x]
= 5 ADD y

Ex.

$(\text{let } y = x+2 \text{ in } x+x)[3/x] = \text{let } y = 3+2 \text{ in } 3+3$ ✓

$(\text{let } y = x+2 \text{ in } x+x)[3/y] = \text{let } y = x+2 \text{ in } x+x$ ✓

y è variabile legata

$(\text{let } y = x+2 \text{ in } x+x)[y/x] = \text{let } y = y+2 \text{ in } y+y$?!

scope di y

Dentro lo scope y è legata

qui y è libera

stiamo andando a legare una variabile libera

Se assumiamo analisi statica (no unbound variable), allora situazioni patologiche sono impossibili

$(let\ x = e_1\ in\ e_2)[v/x]$ → NON possono esserci variabili libere qui

Otteniamo allora la seguente definizione

$(let\ x = e_1\ in\ e_2)[v/x] = let\ x = e_1\ in\ e_2$

$(let\ x = e_1\ in\ e_2)[v/y] = let\ x = e_1[v/y]\ in\ e_2[v/y]$

```
let subst (e: exp) (v: val) (x: identifier) =
    match e with
    ⋮
    | Let (y, e1, e2) -> if y = x then Let (y, e1, e2)
                         else Let (y, subst e v x,
                                      subst e v x)
```

Cosa succede in assenza di analisi statica?

La nozione di sostituzione data scala a linguaggi più espressivi?

App (Lam (x, x), y)

λ-Calcolo

Consideriamo il linguaggio Λ

e ::= x | App e e | Lam x → e

x | e e | fun x → e

x | e e | λx.e

applicazione di funzione

funzioni anonime

```
type exp =
  | Var of identifier
  | App of exp * exp
  | Lam of identifier * exp
```

Programmi = espressioni senza variabili libere

Lam x → x

Lam (x, x)

Il linguaggio Λ è il cuore di ogni linguaggio funzionale

- Ogni espressione di Λ rappresenta una funzione
- Lam costruisce funzioni

  identità Lam x → x
- App applica una funzione ad un argomento

App (Lam f → (Lam x → App f (App f x))) (Lam y → y)

funzione di ordine superiore

- Non ci sono tipi; possiamo applicare una funzione a se stessa

  App (Lam x → x) (Lam x → x)

  → essenza della ricorsione

Fatto. Possiamo codificare SimPL e MiniCaml dentro λ

$\Rightarrow$ capendo λ capiamo aspetti chiave di linguaggi complessi

Come funziona la computazione in λ?

Modello di calcolo per sostituzione

unica operazione primitiva

App (Lam x → e) v → subst e v x (β regola)

```
type val =
  | LamVal of (identifier, exp)
```

```
let is_value (e: exp) : bool =
  match e with
    | Lam (x, e) -> true
    | _ -> false
```

Lam (x, e)

fun x → e

Esercizio. Implementare la β-regola, assumendo di avere subst

(Suggerimento: la β-regola è una relazione tra espressioni

```
let beta (e1 : exp) (e2 . exp) : bool = ...
```

Notare anche che la β-regola è una relazione deterministica: $e \to e_1$ & $e \to e_2$, allora $e_1 = e_2$

```
let beta (e : exp) : exp option
```

Semantica operazionale big-step

$$\frac{}{Lam(x, e) \Rightarrow Lam(x, e)}$$

$$\frac{e_1 \Rightarrow Lam(x, e_1') \quad e_2 \Rightarrow v \quad subst\ e_1'\ v\ x \Rightarrow w}{App(e_1, e_2) \Rightarrow w}$$

(nondeterminismo; β-regola)

NB. (1) Non riduciamo sotto Lam

(2) La β-regola modella passaggio di parametri per valore (call-by-value)

App (Lam (x, e), v) → subst e v x

(λ-notazione: (λx.e) v → e[v/x])

Alla funzione Lam (x, e) passiamo valori (Call-by-Value)

App (Lam (x, e), e') → subst e e' x

(λ-notazione: (λx.e) e' → e[e'/x])

Alla funzione Lam (x, e) passiamo espressioni arbitrarie ovvero riferimenti/"nomi" di valori (Call-by-Name)

Semantica big-step call-by-name

$$\frac{e_1 \Rightarrow Lam(x, e_1') \quad subst\ e_1'\ e_2\ x \Rightarrow v}{App(e_1, e_2) \Rightarrow v}$$

Call-by-Name vs Call-by-Value

① CbV è più efficiente

```
(fun x -> x+x+x+x) (6*7)
```

② CbN calcola solo il necessario

```
(fun x -> 3) (6*7)
(fun x -> 3) loop_forever
```

OCaml usa CbV, essendo più naturale in presenza di: effetti computazionali

Haskell usa una variante efficiente di CbN, detta call-by-need ⇒ computazione lazy

Interprete (Schema generale)

```
let rec eval (e:exp) : val = match e with
    | Var x -> failwith "unbound variable"
    | Lam (x, e) -> LamVal (x, e)
    | App (e1, e2) -> let v1 = eval e1
                          v2 = eval e2      // Non necessario in CbN
                      in
                      match v1 with
                        | LamVal (x, e') -> eval (subst e' v2 x)
                        | _ -> failwith "..."
```

NB. Se siamo certi che non ci siano variabili libere possiamo essere più eleganti:

```
                      ⋮
                      in
                      eval (beta v1 v2)
```

O ancora più sintetic:

```
beta' e v = match e with
  | Lam (x,e;) -> subst e; v x
  | _ -> fail
  :
  | App (e1, e2) -> eval ( beta' (eval e1) (eval e2))
```

Dobbiamo capire come implementare la vera operazione primitiva: la sostituzione

```
let rec subst (e: exp) (v: exp) (x: identifier) : exp =
```

... in CbV is_value v = true

```
    match e with
      | Var y -> if (x = y) then v
                            else Var y
      | App (e1, e2) -> App (subst e1 v x) (subst e2 v x)
      | Lam (y, e1) -> if (x = y) then Lam (y, e1)
                                  else Lam (y, subst e1 v x)
```

$e[v/x]$

$(\lambda y.\, e_1)[v/x] = \lambda y.\, e_1[v/x]$

??

Cosa succede con

subst Lam (x, y) x y

Funzione costante y

= Lam (x, x)

(fun x → y) [x/y]

fun x → y [x/y] = fun x → x

fun x → x

↯ Abbiamo trasformato una funzione costante nella funzione identità] Errore semantico

Semantica let ↯

let y = x in Lam (x, y) ↯ ⇝ Lam (x, x)

Cosa succede se facciamo un renaming?

let y = x in Lam (z, y) ✓ ⇝ Lam (z, x)

Problema. La definizione di sostituzione consente la cattura di variabili libere

→ Rompe le regole di scoping

→ Viola il principio di irrelevanza dei nomi (α-equivalenza)

La semantica di un programma è invariante rispetto alla scelta delle variabili legate

Formalmente: $e \sim_\alpha e'$ $\iff$ e' ottenuto da e rinominando variabili legate

Invarianza $e \sim_\alpha e'$ & $e \Rightarrow v$, allora

$e' \Rightarrow v'$ & $v \sim_\alpha v'$

Soluzione. Data una definizione di: capture-avoiding sostituzione

Def. Definiamo l'insieme $FV(e)$ delle variabili libere di e ricorsivamente

$$FV(\mathrm{Var}\ x) = \{x\}$$
$$FV(\mathrm{App}(e_1, e_2)) = FV(e_1) \cup FV(e_2)$$
$$FV(\mathrm{Lam}(x, e)) = FV(e) \setminus \{x\}$$

Definiamo ora

$$(\mathrm{Lam}(x, e))[v/x] = \mathrm{Lam}(x, e)$$
$$(\mathrm{Lam}(y, e))[v/x] = \mathrm{Lam}(y, e[v/x]) \quad \text{if } y \notin FV(v)$$
$$(\mathrm{Lam}(y, e))[v/x] = \mathrm{Lam}(z, e[z/y][v/x]) \quad \text{if } y \in FV(v) \text{ and } z \text{ variabile nuova}$$
$$\mathrm{Lam}(z, e')[v/x]$$

Es. Calcolare $(\mathrm{fun}\ z \to x)[z/x]$