

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

Prima vedremo un frammento base a cui poi aggiungiamo funzioni:

Linguaggio Sim PL

$e ::= x \mid m \mid b \mid e \text{ binop } e$

$(m \in \mathbb{N}, b \in \{\text{true}, \text{false}\})$

$\mid : f \text{ e Then } e \text{ else } e$

$\mid \text{let } x = e \text{ in } s$

$\text{binop} ::= + \mid * \mid \leq$

$v ::= m \mid b$

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

type **binop** =

- | Plus
- | Times
- | Leq

type **exp** =

- | Var of string
- | Inc of int
- | Bool of bool
- | Binop of binop * exp * exp
- | If of exp * exp * exp
- | Let of string * exp * exp

↳ 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  
  | Inc → true  
  | Bool → true  
  | _ → false
```

Quindi vogliamo

$eval : exp \rightarrow val$

(oppure $eval : exp \rightarrow exp$
tale che
 $is_value (eval e) == true$)

Come possiamo scrivere, in generale, $eval$?

→ semantica operativa

Un interprete è una implementazione della
semantica operativa ~ semantica dinamica

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

Small-step

$$\underline{e \rightarrow e'}$$

passo di computazione singolo

$$e_1 \rightarrow e_1'$$

$$\frac{}{\text{let } x = e_1 \text{ in } e_2 \rightarrow \text{let } x = e_1' \text{ in } e_2}$$

$$\frac{}{\text{let } x = v \text{ in } e \rightarrow e[v/x]}$$

Big-stop

$$\underline{e \Rightarrow v} \quad (\text{oppure } e \Downarrow v)$$

comportamento globale

$$\frac{e_1 \Rightarrow v \quad e_2[v/x] \Rightarrow v'}{\text{let } x = e_1 \text{ in } e_2 \Rightarrow w}$$

Correttezza

$$e \rightarrow^* v$$

sse

$$e \Rightarrow v$$

NB. Solitamente la semantica big-step è nondeterministica

$$\frac{e_1 \Rightarrow \text{Int } m_1 \quad e_2 \Rightarrow \text{Int } m_2}{e_1 \text{ ADD } e_2 \Rightarrow \text{Int } (m_1 + m_2)} \quad \text{Quale eseguite prima?}$$

$$\left(\begin{array}{l} \text{eval (ADD } (e_1, e_2)) = \\ \text{let } v_1 = \text{eval } e_1 \quad \textcircled{1} \\ \quad v_2 = \text{eval } e_2 \quad \textcircled{2} \\ \text{in} \\ \text{primitive ADD } v_1, v_2 \end{array} \right) \text{ vs } \left(\begin{array}{l} \text{eval (ADD } (e_1, e_2)) = \\ \text{let } v_2 = \text{eval } e_1 \quad \textcircled{2} \\ \quad v_1 = \text{eval } e_2 \quad \textcircled{1} \\ \text{in} \\ \text{primitive ADD } v_1, v_2 \end{array} \right)$$

Interprete = determinizzazione \Rightarrow

GESTIONE VARIABILI

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

$$\text{let } x = v \text{ in } e \rightarrow e[v/x]$$

matematicamente comodo
ma poco efficiente

② Ambiente

Un ambiente tiene traccia dei valori assegnati alle variabili

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

$$(\eta, x) \rightarrow (\eta, \eta(x))$$

Interpretare basale su sostituzione

let rec eval (e: exp) : val = tmatch e with

| Int m → Val Int m

| Bool b → Val Bool b

| Var _ → failwith "unbound variable"

| Binop (bop, e₁, e₂) →

let v₁ = eval e₁

v₂ = eval e₂

in primitive_op bop v₁ v₂

} operazione primitiva

| If (e₁, e₂, e₃) → tmatch (eval e₁) with

| Val Bool true → eval e₂

| Val Bool false → eval e₃

| _ → failwith "type error"

} error.
"static:"

| Let (x, e₁, e₂) → let v = eval e₁
in eval (subst e₂ v x)

} sostituzione come
operazioni: primitiva

let primitive-op (bop : binop) (v : val) (v1 : val) : val =
 match bop, v1, v2 with
 | Add, Int a, Int b → Int (a+b)
 | Mult, Int a, Int b → Int (a*b)
 | Leq, Int a, Int b → Bool (a ≤ b)
 | _ failwith "..."
 ↪ chiamata per valore

let subst (e : exp) (v : val) (x : identifier) =

match e with

| Var y → if x=y then v else y

| Int m → Int m

| 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)

| Lot (x, e1, e2) → ???


Ex.


$$(let\ y = x + z\ in\ x + x)\ [3/x] = let\ y = 3 + z\ in\ 3 + 3 \quad \checkmark$$


$$(let\ y = x + z\ in\ x + x)\ [3/y] = let\ y = x + z\ in\ x + x \quad \checkmark$$


y è variabile legata

$$(let\ y = x + z\ in\ x + x)\ [y/x] = let\ y = y + z\ in\ y + y \quad ?!$$


scopo 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*:

$(\text{let } x = e, \text{ in } e_2) [v/x]$ \rightarrow NON possiamo essere: variabili libere qui.

Otteniamo allora la seguente definizione:

$(\text{let } x = e, \text{ in } e_2) [v/x] = \text{let } x = e, \text{ in } e_2$

$(\text{let } x = e, \text{ in } e_2) [v/y] = \text{let } x = e, [v/y] \text{ in } e_2 [v/y]$

$\text{let subst } (e: \text{exp}) (v: \text{val}) (x: \text{identifier}) =$

match e with

⋮

| $\text{let } (y, e_1, e_2) \rightarrow$ if $y = x$ then $\text{let } (y, e_1, e_2)$
 else $\text{let } (y, \text{subst } e \ v \ x, \text{subst } e \ v \ x)$

Cosa succede in assenza di analisi statica?

La nozione di sostituzione data scala a linguaggi più espressivi?

λ -Calcolo

Consideriamo il linguaggio Λ

$$e ::= x \mid \underbrace{App\ e\ e}_{\text{applicazione di funzione}} \mid \underbrace{Lam\ x \rightarrow e}_{\text{funzioni anonime}}$$

$$x \mid e\ e \mid fun\ x \rightarrow e$$

type $exp =$

| Var of identifier
| App of $exp * exp$
| Lam of identifier * exp

Programmi = espressioni senza
variabili libere

Il linguaggio Λ è il ruote di ogni linguaggio funzionale

. Ogni espressione di Λ rappresenta una **funzione**

. **Lam** costruisce funzioni

identità $\text{Lam } x \rightarrow x$

. **App** applica una funzione ad un argomento

$\text{App} (\text{Lam } f \rightarrow (\text{Lam } x \rightarrow \text{App } f (\text{App } f x))) (\text{Lam } y \rightarrow y)$

funzione di ordine superiore

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

$\text{App} (\text{Lam } x \rightarrow x) (\text{Lam } x \rightarrow x)$

→ 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 $(\text{Lam } x \rightarrow e) v \rightarrow \text{subst } e \ v \ x$ (β regola)

type *val* =

| *LamVal* of (*identifior*, *exp*)

| let *is_value* (*c*: *exp*) : bool =
| *match* e with
| *Lam* (*x*, *e*) \rightarrow true
| _ \rightarrow false

Esercizio. Implementare la β -regola, assumendo di avere $subst$
 (Suggerimento: la β -regola è una relazione tra espressioni
 $let\ beta\ (e_1 : exp)\ (e_2 : exp) : bool = \dots$

Notare anche che la β -regola è una relazione
 deterministica: $e \rightarrow e_1$ & $e \rightarrow e_2$, allora $e_1 = e_2$
 $let\ beta\ (e : exp) : exp\ option$

Semantica operativa big-step

$$\frac{\text{Lam}(x, e) \Rightarrow \text{Lam}(x, e) \quad \underbrace{e_1 \Rightarrow \text{Lam}(x, e_1) \quad e_2 \Rightarrow v}_{\text{non determinismo}} \quad \underbrace{subst\ e_1\ v\ x \Rightarrow w}_{\beta\text{-regola}}}{\text{App}(e_1, e_2) \Rightarrow w}$$

NB. (1) Non ridurre sotto Lam

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

$$\text{App}(\text{Lam}(x, e), e') \rightarrow \text{subst } e \ e' \ x$$

(λ -notazione: $(\lambda x. e) e' \rightarrow e[e'/x]$)

Alla funzione $\text{Lam}(x, e)$ passiamo **valor**: (Call-by-Value)

$$\text{App}(\text{Lam}(x, e), e') \rightarrow \text{subst } e \ e' \ x$$

(λ -notazione: $(\lambda x. e) e' \rightarrow e[e'/x]$)

Alla funzione $\text{Lam}(x, e)$ passiamo **espressioni arbitrarie**
ovvero riferimenti / "nomi" di valori (Call-by-Name)

Semantica big-step call-by-name

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

Call-by-Name vs Call-by-Value

① CbV è più efficiente

$$\left(\text{fun } x \rightarrow x + x + x + x \right) (6 * 7)$$

② CbN calcola solo il necessario

$$\left(\text{fun } x \rightarrow 3 \right) (6 * 7)$$

$$\left(\text{fun } x \rightarrow 3 \right) \text{loop-forever}$$

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

Haskell usa una
variante efficiente di
CbN, detta
call-by-need
no computazione
lazy

