Lezione 15-16-17

March 19-21, 2012

Astrazioni di controllo: Costrutti Particolari

- Procedure e Funzioni come parametri: Dove e Perchè
- Decomposition based Programming Methodology
- Problemi di Binding e di Scope: Un esempio
- Trasmissione deep binding: FUI
- Lambda Astrazioni
- Trasmissione shallow binding: FUI
- Astrazioni per supportare Proc. e Funz. Ricorsive: FUI
- Memoization e Tail Recursion: Trasformazioni Interessanti
- Divide and Conquer Programming Methodology
- Funzioni come valori: Higher Order Programming Methodology

Procedure e Funzioni come parametri: Dove e Perchè/1

Decomposition Based Programming Methodology

Dove

- Procedure possono avere come parametri altre procedure o delle funzioni. Lo stesso può essere previsto per le funzioni
- Possiamo astrarre, generalizzare, rispetto ad un'(altra) astrazione (*Vertical Dec.* vs. *Horizontal Dec.*). Ad esempio, in un programma decidiamo:
  - (a) prima di astrarre, in una procedura, P, un'algoritmo di ordinamento su collezioni di valori di un dominio D e
  - (b) dopo, di astrarre, in una funzione, F, l'algoritmo per la relazione di ordine di due arbitrari valori di D.
  - (a+b) è realizzato:

Procedure e Funzioni come parametri: Dove e Perchè/2

Decomposition Based Programming Methodology

Dove:

- (a+b) è realizzato:
  - Usando nel codice C_P della procedura, invocazioni $g(e_1, e_2)$ ogni volta che si abbia necessità di sapere in quale ordine siano i valori di D calcolati da e_1 ed e_2;
  - Estendendo l'intestazione di P in modo tale da includere un parametro funzione g dello stesso tipo T della funzione F;
  - Usando nel programma una definizione per la funzione F accessibile (che abbia nel proprio scope la procedura? -no) alla definizione di P
  - trasmettendo F come parametro funzione ad ogni invocazione di $P(..., T\ F, ...)$

Procedure e Funzioni come parametri: Dove e Perchè/3

Decomposition Based Programming Methodology

- (a+b) è realizzato:
  - Usando nel programma una definizione per la funzione F che sia accessibile all'invocazione della procedura P

Perchè

- F potrebbe essere già stata scritta quando definiamo il programma
- F può essere modificata senza necessità di guardare come sia definita P
- F può essere sostituita in alcune invocazione di P da un'altra funzione che realizza una differente relazione di ordine

Problemi di Binding e di Scope: Un esempio

Example

$$
\begin{array}{l}
\{\ldots \\
\quad \texttt{type } T = \texttt{int} \times \texttt{int} \rightarrow \texttt{bool}; \\
\quad \texttt{int } u\ldots \\
\quad \texttt{function bool } F_1(\texttt{int } x, \texttt{int } y)\{\ldots u\ldots\}; \\
\quad \texttt{procedure } P(T\ g)\{\ldots.\} \\
\quad \texttt{procedure } Q()\{ \\
\qquad \texttt{int } u\ldots \\
\qquad \texttt{function bool } F_2(\texttt{int } x, \texttt{int } y)\{\ldots u\ldots\}; \\
\qquad \ldots P(F_1); \ldots P(F_2); \ldots.\} \\
\quad \ldots \\
\quad P(F_1); \ldots\}
\end{array}
$$

- Il valore denotabile di F_2 sopravvive al suo binding (se abbiamo scope statico)
- Chi è il binding di u? Nelle invocazioni dentro Q, potrebbe essere sempre quello di Q (shallow binding)

Trasmissione Deep Binding, DB (in scope statico)/1

- DB estende in modo naturale lo scope della dichiarazioni alla trasmissione
- Vale anche per procedure ma più interessante su funzioni
- Prima però, vediamo una nuova classe di funzioni: Lambda Astrazioni. Ad esempio $\sharp(x)\ x<=5$

Table12bis – Trasmissione di Valori Funzione
Domini Sintattici
$\texttt{D} ::= \ldots \mid \texttt{Function I}(\texttt{P}_1\ \texttt{I}_1 \ldots \texttt{P}_n\ \texttt{I}_n)\ \texttt{E} \mid \ldots$
$\texttt{E} ::= \ldots \mid \texttt{A} \mid \texttt{Lambda}(\texttt{P}_1\ \texttt{I}_1 \ldots \texttt{P}_n\ \texttt{I}_n)\ \texttt{E} \mid \ldots$

- Lambda Astrazioni non hanno naming: Valori esprimibili, denotabili solo con parametri (e raramente memorizzabili)
- raramente hanno meccanismi per esprimere definizioni ricorsive

Lambda Astrazioni

Table12bis – Trasmissione di Valori Funzione
Domini Sintattici
$\mathtt{D} ::= \ldots \mid \mathtt{Function\ I(P_1\ I_1\ \ldots\ P_n\ I_n)\ E} \mid \ldots$
$\mathtt{E} ::= \ldots \mid \mathtt{A} \mid \mathtt{Lambda(P_1\ I_1\ \ldots\ P_n\ I_n)\ E} \mid \ldots$

Example

$\{\ldots \mathtt{type}\ C(t) = \ldots;\ T_f(t) = t \rightarrow \mathtt{bool};\ T_o(t) = t \times t \rightarrow t; \ldots$
$\mathtt{function}\ C(t)\ Filter(C(t)\ c, T_f(t)\ r)\{\ldots.\};$
$\ldots\{\ldots C(int)v = \ldots;$
$\ldots.Filter(v, \sharp(\mathbf{x})\ \mathbf{x} > \mathbf{5}) \ldots // maggiori\ di\ 5$
$\ldots.Filter(v, \sharp(\mathbf{x})\ \mathbf{x} <= \mathbf{5}) \ldots // minori\ o\ uguali\ a\ 5$
$\ldots\{\ldots \mathtt{type}\ T(t) = C(t) \times T_f(t) \rightarrow C(t);$
$\mathtt{function}\ C(t)\ QuickSort(C(t)\ c, T(t)\ f, T_o(t)o)\{\ldots.\};$
$\ldots.QuickSort(v, Filter, \sharp(\mathbf{x}, \mathbf{y})\ (\mathbf{x} > \mathbf{y})?\mathbf{y} : \mathbf{x})$

Trasmissione Deep Binding, DB (in scope statico)/1

Table12.1 – Deep Binding con Scope Statico

Funzioni Semantiche

$$\mathcal{D}_E[\![\mathtt{D}]\!]_\rho : (\mathtt{Env} \times \mathtt{Store}) \to \mathtt{Env}_\perp$$

$$\mathcal{D}_E[\![\mathtt{Function\ I(Fun\ I_1)\ E}]\!]_\rho =$$
$$\mathtt{Let}\{f = \lambda v\,.\mathtt{Let}\{\rho_1 = \mathtt{bind}(\mathtt{I}_1, \mathtt{F}(v), \rho)\}$$
$$\mathcal{E}[\![\mathtt{E}]\!]_{\rho_1}\}$$
$$\mathtt{bind}(\mathtt{I}, \mathtt{F}(f), \rho)$$

Funzioni Ausiliarie

$$\mathtt{F} : \mathtt{Fun} \to \mathtt{Den}$$

$$\in \mathtt{Fun} : \mathtt{Val} \to \mathtt{TruthV}$$

Trasmissione Deep Binding, DB (in scope statico)/2

Table12.1 – Deep Binding con Scope Statico

Funzioni Semantiche

$\mathcal{E}[\![\mathtt{E}]\!]_\rho : \mathtt{Env} \to \mathtt{State} \to \mathtt{State}_\perp$

$$\mathcal{E}[\![\mathtt{Call\ I(Fun(E))}]\!]_\rho(\mathtt{s}) = \mathtt{Let}\{g = \mathcal{E}[\![\mathtt{E}]\!]_\rho, \mathtt{F}(f) = \rho(\mathtt{I})\} f(g)(\mathtt{s})$$

$$\mathcal{E}[\![\mathtt{Lambda(byValue\ I_1)E}]\!]_\rho(\mathtt{s}) = \lambda\, v\,.\lambda\, s\,.\mathtt{Let}\{(l_1, s_1) = \mathit{allocate}(s)\} \{s_2 = \mathtt{upd}(l_1, v, s_1),\ \rho_1 = \mathtt{bind}(\mathtt{I}_1, l_1, \rho)\} \mathcal{E}[\![\mathtt{E}]\!]_{\rho_1}(s_2)\}$$

Domini Ausiliari

$\mathtt{Den} ::= \mathtt{Loc} + \mathtt{ProcFun} + \mathtt{VL} + \mathtt{Code} + \mathtt{Fun}$

$\mathtt{Fun} ::= \mathtt{State} \to \mathtt{State}$

Trasmissione Shallow Binding, SB (in scope statico)/1

- SB non estende in modo naturale lo scope statico della dichiarazioni alla trasmissione
- Occorre ri-definire le dichiarazioni di funzione (e procedura)
- Quando invocate hanno scope statico, quando trasmesse dinamico
- Il valore denotabile di una funzione è una chiusura

Table12.1 – Shallow Binding con Scope Statico

$$\mathcal{D}_E[\![\texttt{Function I(Fun I}_1\texttt{) E}]\!]_\rho =$$
$$\texttt{Let}\{f = \lambda\delta.\lambda v.\texttt{Let}\{\rho_1 = \texttt{bind}(\texttt{I}_1, v, \delta)\}$$
$$\mathcal{E}[\![\texttt{E}]\!]_{\rho_1}\}$$
$$\texttt{bind}(\texttt{I}, [\texttt{F}(f), \rho], \rho)$$

Funzioni Ausiliarie

$$\texttt{Den} ::= \texttt{Loc} + \texttt{ProcFun} + \texttt{VL} + \texttt{Code} + \texttt{Fun} + [\_, \_]$$
$$\texttt{Fun} ::= \texttt{State} \rightarrow \texttt{State}$$

Trasmissione Shallow Binding, SB (in scope statico)/2

Table12.1 – ShallowBinding con Scope Statico

Funzioni Semantiche

$\mathcal{E}[\![\mathtt{E}]\!]_\rho : \mathtt{Env} \to \mathtt{State} \to \mathtt{State}_\perp$

$$\mathcal{E}[\![\mathtt{Call\ I(Fun(E))}]\!]_\rho(\mathtt{s}) =$$
$$\mathtt{Let}\{[\mathtt{F}(g), \delta_g] = \mathcal{E}[\![\mathtt{E}]\!]_\rho, [\mathtt{F}(f), \delta_f] = \rho(\mathtt{I})\}$$
$$f(\delta_f)([\lambda\delta\,.\,g(\rho), \rho])(\mathtt{s})$$

$$\mathcal{E}[\![\mathtt{Lambda(byValue\ I_1)E}]\!]_\rho(\mathtt{s}) =$$
$$\mathtt{Let}\{h = \lambda\delta\,.\,\lambda v\,.\,\lambda s\,.\,\mathtt{Let}\{(l_1, s_1) = \mathit{allocate}(s)\}$$
$$\{s_2 = \mathtt{upd}(l_1, v, s_1),\ \ \delta_1 = \mathtt{bind}(\mathtt{I}_1, l_1, \delta)\}$$
$$\mathcal{E}[\![\mathtt{E}]\!]_{\delta_1}(s_2)\}\}$$
$$[\mathtt{F}(h), \rho]$$

Trasmissione Shallow Binding, SB (in scope statico)/3

Example

$$
\begin{array}{l}
\{\ldots \texttt{type}\ T = \texttt{int} \times \texttt{int} \rightarrow \texttt{bool};\\
\quad \texttt{int}\ u\ldots\\
\quad \texttt{function bool}\ F_1(\texttt{int}\ x, \texttt{int}\ y)\{\ldots u\ldots\};\\
\quad \texttt{procedure}\ P(T\ g)\{\ldots.\}\\
\quad \texttt{procedure}\ Q()\{\\
\qquad \texttt{int}\ u\ldots\\
\qquad \texttt{function bool}\ F_2(\texttt{int}\ x, \texttt{int}\ y)\{\ldots u\ldots\};\\
\qquad \ldots P(F_1); \ldots P(F_2); \ldots.\}\\
\quad \ldots\\
\quad P(F_1); \ldots\}
\end{array}
$$

- Con SB e scope statico:
  - le funzioni F_1 ed F_2 trasmesse in invocazioni di P, interne a Q, usano l'ambiente di Q per i binding non locali
  - l'ambiente di uso di F_1 ed F_2 è quello che vede Q.
  - Lo stesso avviene per la funzione F_1: l'ambiente è ora quello del blocco esterno

Astrazioni per supportare Proc. e Fun Ricorsive: FUI

- Astrazioni necessarie, anche, per Inductive Programming Methodology (IPM)
- IPM richiede Funzioni Ricorsive
- Funzioni Ricorsive usualmente introdotte combinando: Naming della funzione + Scope includente il corpo stesso

Table12.ter – Funzioni Ricorsive

Funzioni Semantiche

$\mathcal{D}_E[\![\mathtt{D}]\!] : \mathtt{Env} \to \mathtt{Env}_\perp$

$$\mathcal{D}_E[\![\mathtt{Function\ I()E}]\!]_\rho = \mathtt{Y}\ \delta.\mathtt{bind}(\mathtt{I}, \mathtt{F}(\lambda \mathtt{s}.\mathcal{E}[\![\mathtt{E}]\!]_\delta(\mathtt{s})), \delta)$$

- Può essere estesa con parametri (come visto per le procedure)
- Può essere ridefinita per Shallow and Deep binding (come visto per le funzioni)

Astrazioni Ricorsive: Uso (IPM) e Implementazione/1

- Uso: Inductive Programming Methodology, IPM
- IPM: Il problema è risolto utilizzando un algoritmo induttivo
  - Richiede un ordinamento ben fondato sui valori del dominio di definizione (Ogni valore deve avere solo catene finite di predecessori).
  - Il valore calcolato in ogni punto può dipendere solo da valori calcolati su (un numero finito di) predecessori
- Implementazione
  - Stack di AR: tanti AR quanti i predecessori su cui occorre calcolare
  - *Memoization* riduce gli AR
  - *Tail Recursion* ne elimina la necessità

Astrazioni Ricorsive. Implementazione/2 - Memoization

- **Memoization** riduce gli AR
- Una funzione memoized ricorda i valori calcolati in precedenti invocazioni e non li ricalcola,
  - Efficienza
  - Complessità: la memoization della fibonacci ricorsiva rende il calcolo lineare, O(n), invece che esponenziale, $O(2^n)$.
- Tantissime implementazioni diverse (Haskell prevede un meccanismo di memoization)

Astrazioni Ricorsive. Implementazione/3 - Tail Recursion

- **Tail Recursion** elimina la necessità di uno stack di AR
- Una funzione f è tail recursive **sse** il valore calcolato da ogni invocazione ricorsiva di f, nel corpo della sua definizione, è il valore calcolato dalla funzione stessa.
- Poche definizioni di funzione sono tail recursive

Example

$$\texttt{function int } \mathit{fact}(\texttt{int } n)\{ \\ (n == 0)\ ?\ n : n * \mathit{fact}(n-1);\}$$

- ma molte funzioni ricorsive possono essere ridefinite in modo tail recursive.

Astrazioni Ricorsive. Implementazione/4 - Tail Recursion

Example

$$\texttt{function int } \mathit{fact}(\texttt{int } n)\{ \\ (n == 0)\ ?\ n : n * \mathit{fact}(n-1);\}$$

- ma molte funzioni ricorsive possono essere ridefinite in modo tail recursive.

Example

$$\texttt{function int } \mathit{factT}(\texttt{int } n, \texttt{int } r)\{ \\ (n == 0)\ ?\ r : \mathit{factT}(n-1, n * r);\}$$

Astrazioni Ricorsive. Implementazione/5 - Tail Recursion

`function int` $F($`int` $n,$ `int` $r)\{(n == 0)?r : F(n-1, n*r);\}$

- Implementazione di tail recursive.
- una funzione ricorsiva F *riconosciuta o dichiarata Tail Recursive*
  - Calcolo dei parametri: Ad ogni invocazione interna di F (ad esempio, $F(n-1, n*r)$) sono calcolati i parametri attuali nell'AR corrente e disposti nei valori temporanei (stack dei temporanei);
  - Deattivazione AR corrente: Lo AR, C, corrente è deattivato e un nuovo AR, N, è creato ed inizializzato con i link c. statica e dinamica, l'indirizzo di ritorno, l'indirizzo del valore di ritorno, di C.
  - Trasmissione: i valori temporanei di C sono utilizzati nella tramissione e legati nel frame di N ai binding dei parametri di F.
  - Rimozione e Inizializzazione: 1) C è rimosso; 2) il Location Counter di N è inizializzato al codice di fact; 3) N diventa il corrente AR.

Divide and Conquer Programming Methodology/1

- Estende la Programmazione Induttiva
- (1) Il problema è suddiviso in sotto-problemi di tipo correlato:
  - richiedono l'uso delle stesse funzionalità che si stanno definendo e usando per risolvere il problema di partenza
- (2) Consideriamo ciascun sotto-problema:
  - (2.a) Il problema ha immediata soluzione: Terminiamo con tale soluzione
  - (2.b) Iteriamo il processo (1) sul sotto-problema (di non immediata soluzione).

Divide and Conquer Programming Methodology/2

Example

$\{$...type $C(t) = ...$; $T_f(t) = t \rightarrow \texttt{bool}$; $T_o(t) = t \times t \rightarrow \texttt{bool}$; ...
function int $Size(C(t)\ c)\{....\}$; function t $Sel(C(t)\ c)\{....\}$;
function $C(t)$ $AddE(t\ u, C(t)\ c)\{....\}$;
function $C(t)$ $Append(C(t)\ c_1, C(t)\ c_2)\{....\}$;
function $C(t)$ $Filter(C(t)\ c, T_f(t)\ r)\{....\}$;
...
$\{...C(int)v = ...; .....$
$\{$...type $T_z(t) = C(t) \rightarrow \texttt{int}$; $T_a(t) = C(t) \times C(t) \rightarrow C(t)$;
$T_e(t) = t \times C(t) \rightarrow C(t)$; $T_s(t) = C(t) \rightarrow t$;
function $C(t)\ QuickS(C(t)c, T_s(t)s, T_z(t)z, T_a(t)a, T_e(t)e, T_f(t)f, T_o(t)o)\{$
if $(z(c) < 2)$ return c;
$\{t\ u = s(c); C(t)\ gt = f(c, \sharp(x) \rightarrow o(u, x))$;
$C(t)\ lt = f(c, \sharp(x) \rightarrow o(x, u))$;
return $a(QuickS(lt, s, z, a, e, f, o), e(u, QuickS(gt, s, z, a, e, f, o)))$;
$\}$;
$....QuickS(v, Sel, Size, Append, AddE, Filter, \sharp(\mathbf{x}, \mathbf{y})\ (\mathbf{x} > \mathbf{y})?\mathbf{y} : \mathbf{x})$

Divide and Conquer Programming Methodology: Uso di interfacce/3

- occorre sempre trasmettere tutte le operazioni del tipo $C(t)$?
- No se abbiamo tipi classe, Haskell (interfaccia in Java)
- `Class` $Ord(t) => C(t)$ `where`... // specifica operazioni

Example

$\{$....`type` $T_o(t) = t \times t \rightarrow$`bool`;
...$\{$...$C(int)v = ...$;
$\{$...
`function` $C(t)$ $QuickS(C(t)c, T_o(t)o)\{$
`if`$(Size(c) < 2)$ `return` c;
$\{t\ u = Sel(c); C(t)\ gt = Filter(c, \sharp(x) \rightarrow o(u, x))$;
$C(t)\ lt = Filter(c, \sharp(x) \rightarrow o(x, u))$;
`return` $Append(QuickS(lt, o), AddE(c, QuickS(gt, o)))$;
$\}$;
....$QuickS(v, \sharp(\mathbf{x}, \mathbf{y})\ (\mathbf{x} > \mathbf{y})?\mathbf{y} : \mathbf{x})$

Funzioni come Valori: HOP

Full Higher Order Programming Language

- Le funzioni sono valori *first class*: i.e. di base per la programmazione nel linguaggio
- Valori funzione usati come parametri attuali nelle invocazioni
- Valori funzione usati come valore calcolato dalle invocazioni
- Vantaggi: Siamo all'apice dell'espressività di un linguaggio
  - Il programma **calcola** (durante l'esecuzione) le funzioni con cui deve calcolare
  - i costrutti del linguaggio sono descritti da funzioni semantiche e queste sono funzioni calcolabili
  - il programma può (all'occorrenza) introdurre alcune di tali funzioni calcolabili
    - come nuovi tipi di dato
    - come nuovi tipi di costrutti di controllo della composizione.

Higher Order Programming: Formalizzazione/2

La presenza di valori funzione calcolati non crea problemi alla formalizzazione

- Avevamo già ampliato il dominio dei valori esprimibili includendo le funzioni:
  - quando utilizzate come parametri attuali in una trasmissione di funzione (o procedura)
  - quando espresse con Lambda Astrazioni
- Ora potremmo decidere di ampliare il dominio dei valori memorizzabili per includere funzioni che possono anche essere assegnate a variabili una volta calcolate da un'invocazione (come nell'esempio del lucido che segue)

Higher Order Programming: Formalizzazione/3

- ..ampliare il dominio dei valori memorizzabili per includere funzioni assegnabili a variabili ...

Table12.quater – Funzioni come Memorizzabili

Funzioni Semantiche

$\mathcal{E}[\![\mathtt{E}]\!]_\rho : \mathtt{Env} \rightarrow \mathtt{State} \rightarrow \mathtt{State}_\perp$

$\mathcal{E}[\![\mathtt{Call\ I(...)}]\!]_\rho(\mathtt{s}) =$

$\mathtt{Let}\{..., f = \mathcal{Q}(\mathtt{I})(\rho)(\mathtt{s})...\}f(...)(\mathtt{s})...$

$\mathcal{Q}(\mathtt{I})(\rho)(\mathtt{s}) = if(\rho(i) \in \mathtt{Fun}, \mathtt{Let}\{\mathtt{F}(f) = \rho(i)\}f, \mathtt{Let}\{\mathtt{F}_M(f) = \mathtt{s}(\rho(i))\}f$

Funzioni Ausiliarie

$\mathtt{F_M} : \mathtt{Fun} \rightarrow \mathtt{Mem}$

Domini Ausiliari

$\mathtt{Mem} ::= \mathtt{VL} + \mathtt{Fun}$

$\mathtt{Fun} ::= \mathtt{State} \rightarrow \mathtt{State}$

Higher Order Programming: Implementazione/3

La presenza di valori funzione calcolati non crea problemi

- Apparentemente, niente da aggiungere allo stack di AR
- Ma, usiamolo per calcolare:

Example

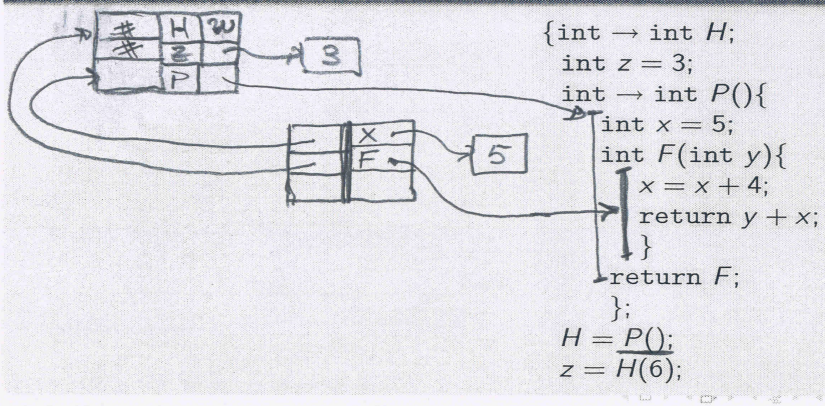
```
{int → int H;
 int z = 3;
 int → int P(){
    int x = 5;
    int F(int y){
       x = x + 4;
       return y + x;
       }
     return F;
     };
 H = P();
 z = H(6);
```

Higher Order Programming: Implementazione/3

La presenza di valori funzione calcolati non crea problemi

- Apparentemente, niente da aggiungere allo stack di AR
- Ma, usiamolo per calcolare:

Example

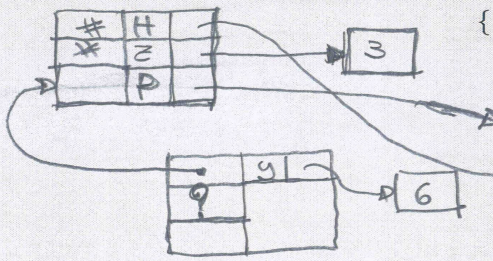


Higher Order Programming: Implementazione/3

La presenza di valori funzione calcolati non crea problemi

- Apparentemente, niente da aggiungere allo stack di AR
- Ma, usiamolo per calcolare:

Example




```
{int → int H;
 int z = 3;
 int → int P(){
   int x = 5;
   int F(int y){
     x = x + 4;
     return y + x;
   }
   return F;
 };
 H = P();
 z = H(6);
```

Higher Order Programming: Valutazione/1

La presenza di valori funzione calcolati non crea problemi alla formalizzazione data: Applichiamo la semantica data

Example

Calcolo blocco

ambiente − memoria $\rho_0 \circ \rho_0', \sigma$:

$\rho_0(H) = l_1 \quad \sigma(l_1) = \omega$

$\rho_0(z) = l_2 \quad \sigma(l_2) = 3$

$\rho_0'(P) = \lambda \mathtt{s} . \mathcal{C}[\![C_P]\!]_{\rho_0}(\mathtt{s})$ − *dove* C_P = *corpo di* P

Calcolo di $H = P()$

Invocazione $P() : \mathcal{C}[\![C_P]\!]_{\rho_0}(\sigma)$

ambiente − memoria $\rho_1, \sigma : \rho_1 = \rho_1'' \circ \rho_1' \circ \rho_0$

$\rho_1'(x) = l_3 \quad \sigma(l_3) = 5$

$\rho_1''(F) = \lambda y . \lambda \mathtt{s} . \mathcal{E}[\![C_F]\!]_{\rho_1' \circ \rho_0}$ − *dove* C_F = *corpo di* F

`return` F *restituisce* $\rho_1''(F)$

$P()$ *calcola* $\lambda y . \lambda \mathtt{s} . \mathcal{E}[\![C_F]\!]_{\rho_1' \circ \rho_0}$

Higher Order Programming: Valutazione/2

Example

Calcolo blocco

ambiente − memoria $\rho_0 \circ \rho_0', \sigma$:

$\rho_0(H) = l_1 \quad \sigma(l_1) = \omega$

$\rho_0(z) = l_2 \quad \sigma(l_2) = 3$

$\rho_1(P) = \lambda \mathtt{s} \,.\, \mathcal{C}[\![C_P]\!]_{\rho_0}(\mathtt{s})$ − −*dove* C_P *sia il corpo di* P

Calcolo di $H = P()$ *risulta in* $\sigma(l_1) = \lambda y \,.\, \lambda \mathtt{s} \,.\, \mathcal{E}[\![C_F]\!]_{\rho_1' \circ \rho_0}$

$\rho_1'(x) = l_3 \quad \sigma(l_3) = 5$

Calcolo di $z = H(6)$

Calcolo di $H(6)$

Calcolo di $H : \sigma(\rho_0(H)) = \lambda y \,.\, \lambda \mathtt{s} \,.\, \mathcal{E}[\![C_F]\!]_{\rho_1' \circ \rho_0}$

Invocazione di $H(6)$

ambiente − memoria $\rho_2, \sigma : \rho_2 = \rho_2' \circ \rho_1' \circ \rho_0$

$\rho_2'(y) = l_4 \quad \sigma(l_4) = 6$

$x{+}4$ *valuta* $\sigma(\rho_2(x)){+}4 = \sigma(l_3){+}4 = \ 9$

$x = x{+}4$ *risulta in* $\sigma(l_3) = 9$

Higher Order Programming: Valutazione/3

Example

Calcolo blocco

ambiente − memoria ρ_0, σ :

$\rho_0(H) = l_1 \quad \sigma(l_1) = \omega$

$\rho_0(z) = l_2 \quad \sigma(l_2) = 3$

$\rho_1(P) = \lambda \mathtt{s}\,.\,\mathcal{C}[\![C_P]\!]_{\rho_0}(\mathtt{s})$ − −*dove* C_P *sia il corpo di* P

Calcolo di $H = P()$ *risulta in* $\sigma(l_1) = \lambda y\,.\,\lambda \mathtt{s}\,.\,\mathcal{E}[\![C_F]\!]_{\rho_1' \circ \rho_0}$

$\rho_1'(x) = l_3 \quad \ldots$

Calcolo di $z = H(6)$

Calcolo di $H(6)$

ambiente − memoria $\rho_2, \sigma : \rho_2 = \rho_2' \circ \rho_1' \circ \rho_0$

$\rho_2'(y) = l_4 \quad \sigma(l_4) = 6$

$x = x{+}4$ *risulta in* $\sigma(l_3) = 9$

$y{+}x$ *valuta* $\sigma(\rho_2(y)) + \sigma(\rho_2(x)) = \sigma(l_4) + \sigma(l_3) = 15$

`return` *restituisce* 15

Calcolo di $z = H(6)$ *risulta in* $\sigma(l_2) = 15$

Higher Order Programming: Implementazione/4

La presenza di valori funzione calcolati non crea problemi alla formalizzazione

- Ma l'implementazione con stack di AR non opera correttamente
- Due soluzioni:
  - Currying + Lambda Lifting (statico)
  - Stack con AR-Retention

HOS: Currying+Lambda Lifting (statico)/1

- Lambda Lifiting:
  - non locali passati come parametri:
    Esempio: $\texttt{int}\ F(\texttt{int}\ x, \texttt{int}\ y)\{...\}$
  - non locali a valori modificabili: trasmissione per reference
    Esempio: $\texttt{int}\ F(\texttt{ref int}\ x, \texttt{int}\ y)\{...\}$
- Currying:
  $F_k : A_1 \times ... \times A_k \rightarrow B$ equivale a $F_k : A_1 \rightarrow (... \rightarrow (A_k \rightarrow B)...)$
  - Invocazione (Valutazione) parziale delle funzioni:
    Esempio: con scope statico
    $\texttt{return}\ F$; è sostituita da $\texttt{return}\ F(x)$;

HOS: Currying+Lambda Lifting (statico)/2

Example

```
{int → int H;
 int z = 3;
 int → int P(){
    int x = 5;
    int F(int y){
       x = x + 4;
       return y + x;
       }
    return F;
    };
 H = P();
 z = H(6);
```

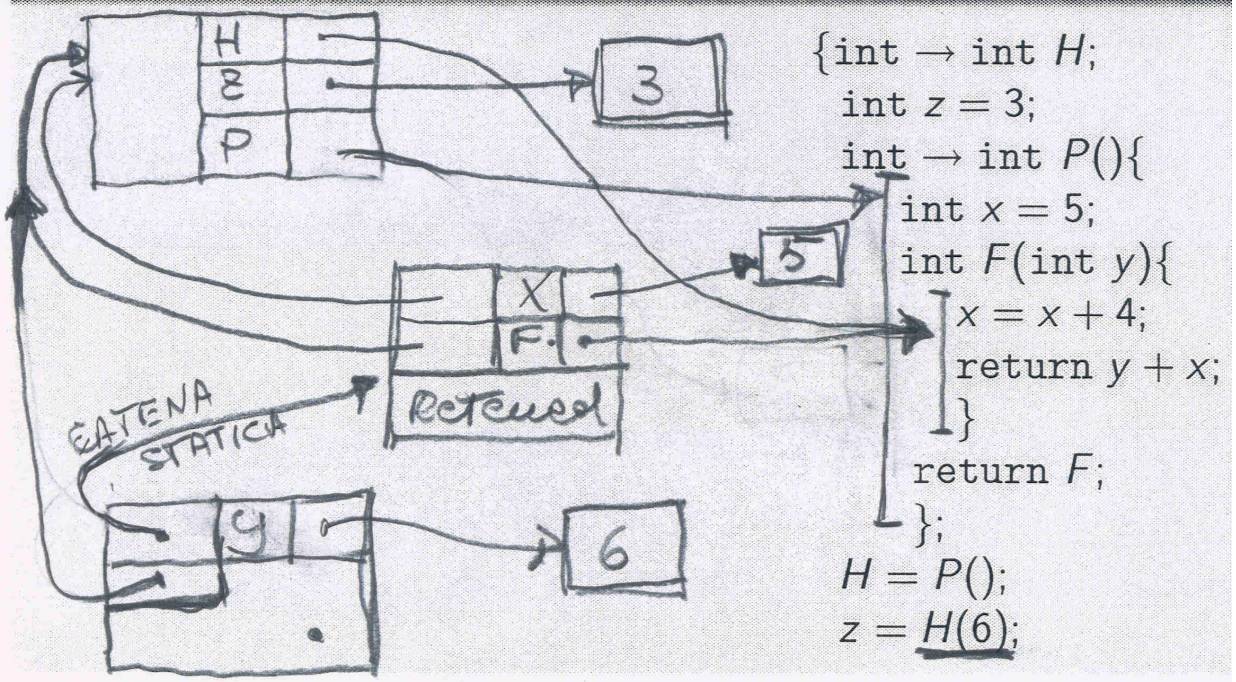
```
{int → int H;
 int z = 3;
 int → int P(){
    int x = 5;
    int F(ref int x, int y){
       x = x + 4;
       return y + x;
       }
    return F(x);
    };
 H = P();
 z = H(6);
```

HOS: Stack con AR Retention

L'implementazione gli AR **retained** che sono AR che dovrebbere essere popped ma sono mantenuti perchè altri AR potrebbero utilizzarli nella catena statica dei frame

nella catena statica dei frame.

Example



Lezione 15-16-17