# Lezione 9-10

March 5, 2012

# Espressioni: Formalizzazione, Uso, Implementazione

- Espressioni: Trasparenza Referenziale e Side Effects
- Espressioni Divergenti: operatori (funzioni) stretti e non stretti
- Valori Strutturati, Costruttori Lazy, Valutazioni Eager e Lazy
- Valori Finitari e Valori Infiniti
- Allocazione e Memoria finite: Formalizzazione
- Valori modificabili e assegnamento
- Formalizzazione degli argomenti trattati

# Espressioni: Trasparenza Referenziale e Side Effects/1

- Espressioni con Trasparenza Referenziale

  Possiamo rimpiazzare l'espressione con il suo valore senza che ciò cambi il comportamento del programma.
  - Linguaggi Funzionali Puri
  - Funzione Semantica

    $$\mathcal{E} : \texttt{E} \rightarrow \texttt{Env} \rightarrow \texttt{State} \rightarrow \texttt{Val}$$

**Example**

L'espressione $e_1 + e - e_1$ può essere rimpiazzata dall'espressione $e$ sempre allorchè:
1) $e_1 \in \texttt{Val}$ (la sua computazione termini e calcoli un valore);
2) gli operatori $+$ e $-$ siano gli operatori di somma e sottrazione.

# Espressioni: Trasparenza Referenziale e Side Effects/2

- Espressioni con Side Effects

  Le espressioni non esprimono solo valori ma possono nascondere cambiamenti dello stato

  - Linguaggi Procedurali
  - Funzione Semantica

    $$\mathcal{E} : \mathtt{E} \to \mathtt{Env} \to \mathtt{State} \to (\mathtt{Val} \times \mathtt{State})$$

**Example**

L'espressione $e_1 + e - e_1$ non può essere rimpiazzata dall'espressione $e$
anche quando:
1) $e_1 \in \mathtt{Val}$ (la sua computazione termini e calcoli un valore);
2) gli operatori $+$ e $-$ siano gli operatori di somma e sottrazione.
Allo scopo si consideri l'espressione $e_1 \equiv x = y$.

# Espressioni: Uso

- **Uso**
  - Esprimere valori ottenibile attraverso la composizione di applicazioni di operatori primitivi e non su valori e riferimenti a valori (e.g. naming).
  - Esprimere algoritmi che utilizzano (anche complicate) manipolazione di valori (modificabili e non)
  - Nei Linguaggi Funzionali tutte le funzioni calcolabili sono espresse attraverso programmi che implementano solo algoritmi formulati in tale modo

# Espressioni: Implementazione

- Decomposizione (compile time) di espressione in sequenze ordinate di operazioni elementari
  - **un solo operatore** è coinvolto in ogni operazione della sequenza
  - la sequenza di operazioni è gestita in forma indiretta dalla rappresentazione (in genere, sintassi astratta):
    - Notazione infissa. Abstract Tree e visita Depth-First: $3 * x + 2$ diventa $tree(tree(3, *, x), +, 2)$.
    - Notazione prefissa. Inner Sequencing: $3 * x + 2$ diventa $[+, *, 3, x, 2]$.
    - Notazione postfissa. Sequencing: $3 * x + 2$ diventa $[3, x, *, 2, +]$ – reversed è perfetta per stack degli intermedi
- Stack degli Intermedi: Decomposizione delle applicazioni in una sequenza ordinata; applicazioni degli operatori e generazione di risultati intermedi formanti nuovi operandi per le applicazioni ancora sospese
- Stack AR: applicazioni di astrazioni funzionali

# Espressioni: Divergenti

- Le espressioni non calcolano solo valori

**Example**

Cosa calcola l'espressione C seguente?

$$0 * fact(-3)$$

allorchè *fact* sia la seguente procedura C:

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

- Possono condurre a calcoli divergenti, non terminanti:
  - $\mathcal{E} : \texttt{E} \to \texttt{Env} \to \texttt{State} \to \texttt{Val}_\perp$
  - $\mathcal{E} : \texttt{E} \to \texttt{Env} \to \texttt{State} \to (\texttt{Val} \times \texttt{State})_\perp$
- Questo fatto invece di essere un limite per gli L.P. ne può aumentare l'espressività.

# Espressioni: Operatori (funzioni) Stretti e non Stretti/1

- Questo fatto invece di essere un limite per gli L.P. ne può aumentare l'espressività.

**Example**

Cosa calcolano le seguenti due espressioni C?

$$(x < 0)||(fact(x) > 0)$$
$$(x < 0)|(fact(x) > 0)$$

- Possono condurre a calcoli divergenti, non terminanti:
  - $\mathcal{E} : \mathtt{E} \rightarrow \mathtt{Env} \rightarrow \mathtt{State} \rightarrow \mathtt{Val}_{\perp}$
  - $\mathcal{E} : \mathtt{E} \rightarrow \mathtt{Env} \rightarrow \mathtt{State} \rightarrow (\mathtt{Val} \times \mathtt{State})_{\perp}$
- Operatori
  - **non stretti**: $||, \&\&$.
  - **stretti**: $|$
- Semantica
- Implementazione

# Espressioni: Operatori (funzioni) Stretti e non Stretti/2

Semantica di op:$\texttt{Val}^k \rightarrow \texttt{Val}$:

- Dominio Val è esteso: $\texttt{Val}_\perp$
- Stretti. Valutazione di tutti gli argomenti: $\text{op}_{\perp_S}$

$$\text{op}_{\perp_S}(e_1, ..., e_k) = \begin{cases} \text{op}(e_1, ..., e_k) & \text{sse } (\forall i \in [1..k]) e_i \in \texttt{Val} \\ \perp_{\texttt{Val}} & \text{otherwise} \end{cases}$$

- Non Stretti. Valutazione dei soli argomenti necessari al calcolo: $\text{op}_{\perp_{NS}}$
  $e_i \in \texttt{Val}\ (\forall i \in [1..k]) \Rightarrow \text{op}_{\perp_{NS}}(e_1, ..., e_k) = \text{op}(e_1, ..., e_k)$

**Example**

Sia op $= \star$. Allora
$\star_{\perp_S}$ esteso nell'unico modo possibile.
$\star_{\perp_{NS}}$ potrebbe essere esteso in uno dei seguenti:
$\star_{\perp_{NS}} = \lambda(\texttt{x}, \texttt{y}).\texttt{if } (\texttt{x} == 0) \texttt{ then } 0 \texttt{ else } \star_{\perp_S}(x, y)$
$\star_{\perp_{NS}} = \lambda(\texttt{x}, \texttt{y}).\texttt{if } (\texttt{x} == 0 \,||\, \texttt{y} == 0) \texttt{ then } 0 \texttt{ else } \star_{\perp_S}(x, y)$

# Espressioni: Operatori (funzioni) Stretti e non Stretti/3

Implementazione

- Operatori Primitivi:
  - Non Stretti: Valutazione dei soli argomenti necessari al calcolo (short circuit)
  - Stretti: Valutazione di tutti gli argomenti
- Funzioni:
  - Non Strette: Valutazione dei soli argomenti necessari al calcolo (call by need)
  - Strette: Valutazione di tutti gli argomenti (call by value)

**Example**

Si consideri la seguente espressione, Haskell, `h 3 (f 5)`, allorchè `h` e `f` siano:

```
h = \x y -> if (x\=0) then x else y
f n = f(n+1)
```

In Haskell essa è valutata al valore 3.
Cosa possiamo dire di essa in OCaml, C o in Java?

# Espressioni: Funzioni Strette e non Strett3/4

Funzioni non strette

- Molto utili nella programmazione
- Ovviamente non sempre calcolabili

## Example

Una funzione non stretta e non calcolabile è *Halting*. Questa funzione potrebbe essere descritta in questo modo:

- naming. halting
- tipo. $\texttt{Val}_\perp \longrightarrow \texttt{Bool}$
- comportamento. Applicata ad un'espressione calcola `true` se l'espressione calcola un valore, ovvero un membro di in `Val`. Calcola `false` se l'espressione è divergente.
- uso. Applicata ad un'espressione `e(e')` dove e calcoli una funzione di tipo $\texttt{Val} \longrightarrow \texttt{Val}$ ed `e'` calcoli un valore di tipo `Val`, Halting ci dice se l'espressione è divergente..

# Espressioni: Eager vs. Lazy Evaluation/1

- Valori Strutturati estendono ulteriormente la nozione di *calcolo non stretto*
  - costruttori di valori
  - operatori per la selezione (e modifica, se modificabili) componenti
- Lazy. Costruttori: Valutazione argomenti solo se necessaria
- Eager. Valutazione di tutti gli argomenti sempre e per tutte le applicazioni

**Example**

Si consideri l'espressione, Haskell, `g [4,(f 5)]`, per `h` e `f`:

```
g u = if ((hd u)==0) then 3 else 7
f n = f(n+1)
```

In Haskell essa è valutata al valore 7.

Cosa possiamo dire di essa in OCaml, C o in Java?

# Espressioni: Lazy Evaluation/2

- Valori Strutturati estendono ulteriormente la nozione di *calcolo non stretto* nel *calcolo Lazy*
- Uso
  - Valori strutturati contengono chiusure per applicazioni sospese
  - Le chiusure sono usate anche come "valori memorizzabili" di componenti delle strutture
  - Valori Infiniti (approssimabili finitamente)
  - Valori Finitari Infiniti (strutture infinite rappresentabili finitamente)
- Implementazione: tante, diverse, estendono la call by-need

# Espressioni: Lazy Evaluation/3

## Example

Valori Infiniti (approssimabili finitamente):

```
nat n = n:nat(n+1)
naturali = nat 0
v = fst 3 naturali
```

In Haskell le tre espressioni calcolano una funzione, la lista di tutti i naturali, la lista dei primi 3 naturali.
Possiamo esprimere tali valori in Caml, C, Java?

## Example

Valori Finitari Infiniti (rappresentabili finitamente):

`data Tree a = T(a,Tree a)` - tipo polimorfo di Haskell

`treeM = T(3,treeM)` - un valore Haskell

treeM esprime un albero infinito rapresentabile finitamente (con puntatori!!)
Cosa possiamo dire di essa in Caml, C o in Java?

# Espressioni: Denotabili

Le espressioni `E` che calcolano valori modificabili hanno due significati diversi.

- Valore Memorizzabile. Accede il valore esprimibile associato: `Val(E)`
- Valore Denotabile. Accede il valore (per modificarlo): `Den(E)`

**Example**

In C (e nei linguaggi procedurali) la seguente dichiarazione:

```
int x;
```

Introduce un naming, l'identificatore `x`, per un valore modificabile.Il valore modificabile introdotto è una variabile intera: A livello semantico (sintassi astratta), indicheremo tale valore con `Den(x)`. Il valore della variabile è un valore memorizzabile che indicheremo, a livello semantico (s.a.) con `Val(x)`

# Espressioni: con S.E./1

Le espressioni possono limitarsi a calcolare un valore (trasparenza referenziale) oppure modificano anche lo stato generando *Side Effects*.

**Table9 – Semantica delle Espressioni con S.E. – 1**

**Domini Sintattici**

$\texttt{D} ::= \ldots \mid \texttt{Var I;} \mid \texttt{Var I} = \texttt{E;} \mid \ldots$

$\texttt{E} ::= \texttt{Val(E)} \mid \texttt{Den(E)} \mid \texttt{VL} \mid op_k(\texttt{E}_1 \ldots \texttt{E}_k) \mid \texttt{E} = \texttt{E}$

**Domini Semantici**

$\texttt{Env}, \rho, \delta \equiv \ldots$

$\texttt{Store} \equiv (\texttt{Loc} \times (\texttt{Loc} \rightarrow \texttt{Mem}_\perp))$ (*store con allocazione*)

$\texttt{Store}_\perp, \texttt{s}, \texttt{r} \equiv \texttt{Store} + \{\perp_S\}$ (*store finito*)

# Memoria Finita: Operazioni

$$\mathtt{Store} \equiv (\mathtt{Loc} \times (\mathtt{Loc} \to \mathtt{Mem}_\perp))$$
$$\mathtt{Store}_\perp, \mathtt{s}, \mathtt{r} \equiv \mathtt{Store} + \{\perp_S\}$$

$$\mathtt{upd} : \mathtt{Loc} \times \mathtt{Mem}_\perp \times \mathtt{Store}_\perp \to \mathtt{Store}_\perp$$
$$\mathtt{upd}(l, m, (L, u))$$
$$\equiv \mathtt{if}((l \in [0, L]), \lambda v\,.\,\mathtt{if}((v\ \mathtt{eq}\ l), m, u(l)), \perp_S)$$
$$\mathtt{look} : \mathtt{Loc} \times \mathtt{Store}_\perp \to \mathtt{Mem}_\perp$$
$$\mathtt{look}(l, (L, u)) \equiv \mathtt{if}((l \in [0, L]), u(l), \perp_M)$$
$$\mathtt{allocate} : \mathtt{Store}_\perp \to (\mathtt{Loc} \times \mathtt{Store}_\perp)$$
$$\mathtt{allocate}_k((L, u))$$
$$\equiv \mathtt{if}((L > k), \perp_{LS}, (L, (L + 1, upd(L, \perp_M, u))))$$
$$\perp_S : \mathtt{Store}_\perp$$
$$\perp_S \equiv (\mathtt{Y}f\,.\,\lambda \mathtt{x}\,.\,f(\mathtt{x}))(u), \qquad con\ u \in \mathtt{Store}$$

# Espressioni: con S.E./2 - Dichiarazioni

**Funzioni Semantiche**

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

$$\begin{aligned}
&\mathcal{D}[\![\mathtt{Var\ I;}]\!]_\rho(s) \\
&\quad = \mathtt{Let}\{(l, s_l) = \mathit{allocate}(s)\}\ (\mathtt{bind}(\mathtt{I}, l, \rho), s_l) \\
&\mathcal{D}[\![\mathtt{Var\ I = E;}]\!]_\rho \qquad\qquad\qquad\qquad - \mathit{compile\ time} \\
&\quad = \mathtt{Let}\{(l, s_l) = \mathit{allocate}(s)\} \\
&\qquad\qquad \{(v_e, s_e) = \mathcal{E}[\![\mathtt{E}]\!]_\rho(s_l)\} \\
&\qquad\qquad \{s_m = \mathtt{upd}(l, v_e, s_l)\}\ (\mathtt{bind}(\mathtt{I}, l, \rho), s_m)
\end{aligned}$$

# Espressioni: con S.E./3 - Valori

**Funzioni Semantiche**

$$\mathcal{E}[\![\mathtt{E}]\!]_\rho : \mathtt{Store} \to (\mathtt{Val}_\perp \times \mathtt{Store}_\perp)$$

$$\mathcal{E}[\![\mathtt{Val(E)}]\!]_\rho(s) = \begin{cases} (\mathtt{MV}(\mathtt{s}(\rho(\mathtt{E}))), s) & \text{if } \rho(\mathtt{E}) \in \mathtt{Loc} \\ (\mathtt{DV}(\mathtt{E}), s) & \text{otherwise} \end{cases}$$

$$\mathcal{E}[\![\mathtt{Den(E)}]\!]_\rho(s) = \begin{cases} (\rho(\mathtt{E}), s) & \text{if } \rho(\mathtt{E}) \in \mathtt{Loc} \\ (\perp_D, s) & \text{otherwise} \end{cases}$$

$$\mathcal{E}[\![\mathtt{VL}]\!]_\rho(s) = (\mathtt{IntoVal}(\mathtt{VL}), s)$$

# Espressioni: con S.E./4 - Operatori-1

Operatori che propagano S.E. ma non lo generano.

**Funzioni Semantiche**

$$\mathcal{E}[\![\mathtt{E}]\!]_\rho : \mathtt{Store} \to (\mathtt{Val}_\perp \times \mathtt{Store}_\perp)$$

$$\mathcal{E}[\![op_k(\mathtt{E}_1 ... \mathtt{E}_k)]\!]_\rho(s) \qquad (\textit{stretti})$$
$$= \mathtt{Let}\{(v_1, s_1) = \mathcal{E}[\![\mathtt{E}_1]\!]_\rho(s)\}$$
$$\ldots$$
$$\{(v_k, s_k) = \mathcal{E}[\![\mathtt{E}_k]\!]_\rho(s_{k-1})\}\ (op_{\perp_S}(v_1 ... v_k), s_k)$$

$$\mathcal{E}[\![op_k(\mathtt{E}_1 ... \mathtt{E}_k)]\!]_\rho(s) \qquad (\textit{non stretti})$$

# Espressioni: con S.E./4 - Operatori-2

Operatori che propagano S.E. ma non lo generano.

$$\mathcal{E}[\![op_k(\mathrm{E}_1 ... \mathrm{E}_k)]\!]_\rho(s) \qquad (\textit{non stretti})$$
$$= \mathtt{Let}\{(v_{i_1}, s_{i_1}) = \mathcal{E}[\![\mathrm{E}_{i_1}]\!]_\rho(s)$$
$$...$$
$$\{(v_{i_h}, s_{i_h}) = \mathcal{E}[\![\mathrm{E}_{i_h}]\!]_\rho(s_{i_{h-1}})\}\ (op_{\perp_{NS}}(\overline{v}_1 ... \overline{v}_k), s_{i_h})$$

$$\textit{where}: (i_1 \neq ... \neq i_h \in [1..k])\ \textit{and}\ (\overline{v}_{i_j} \equiv\ v_{i_j}(\forall j \in [1..h]))$$

**Example**

Un operatore non stretto con il seguente comportamento:
let op = fun x y z w $\rightarrow$
if (w=0) then 0 else if (w>y) then y else if ...
quali argomenti valuta, in quale ordine, e ciascun argomento in quale stato?

# Espressioni: con S.E./5 - Operatori-3

Operatori che generano S.E.

**Funzioni Semantiche**

$$\mathcal{E}[\![\mathtt{E}]\!]_\rho : \mathtt{Store} \to (\mathtt{Val}_\perp \times \mathtt{Store}_\perp)$$

$$\begin{aligned}&\mathcal{E}[\![\mathtt{E}_l = \mathtt{E}_r]\!]_\rho(s)\\ &\quad = \mathtt{Let}\{(v_1, s_1) = \mathcal{E}[\![\mathtt{E}_r]\!]_\rho(s)\}\\ &\qquad\quad \{(l_2, s_2) = \mathcal{E}[\![\mathtt{E}_l]\!]_\rho(s_1)\}\ (v_1, \mathtt{upd}(l_2, \mathtt{VM}(v_1), s_2)\end{aligned}$$

**Funzioni Ausiliarie**

$\mathtt{L} : \mathtt{Loc} \to \mathtt{Den}$ $\qquad$ (*iniettore*, *i.e.costruttore*)

$\mathtt{DV} : \mathtt{Den} \to \mathtt{Val}$

$\mathtt{VM} : \mathtt{Val} \to \mathtt{Mem}$