Linguaggi di Programmazione avanzati
Linguaggi funzionali

Simone Martini

Dipartimento di Scienze dell'Informazione
Università di Bologna
Italy

A.a. 2005-2006

university-logo

Outline

university-logo

Computazioni senza stato

- Modello imperativo
  - Computazione è trasformazione dello stato, fissato un ambiente
  - $[\![P]\!] : \mathit{Env} \times \mathit{State} \rightarrow \mathit{State}$
  - Costrutto cruciale: assegnamento
  - Gli effetti collaterali sono la regola, non l'eccezione
  - Controllo: iterazione
- Modello funzionale (puro)
  - Computazione è riscrittura di espressioni
  - $[\![P]\!] : \mathit{Env} \rightarrow \mathit{Value}$
  - Costrutto cruciale: definizione di espressioni (con parametri)
  - Gli effetti collaterali non esistono
  - Controllo: ricorsione

university-logo

Espressioni e funzioni

Sia $f(x) = x^2$ la funzione che associa ad x il suo quadrato. Sia ora $x = 2$: ne segue che $f(x) = 4$.

- In stile C: `int f (int x) {return x*x;}`
- `f(2);`
- `val quattro = f(2);`
- `val` modifica solo l'ambiente
- Specifica sintassi per un'espressione che denota una funzione:
  `val f = fn x => x*x;`
  `return` sottinteso (tutto il corpo è "ritornato")
- `(fn y => y+1) (6);`

university-logo

Le funzioni sono espressioni qualsiasi

- Notazione:
  1. chiamata di funzione: `f 2`
  2. `g a1 a2 ... ak` sta per `(...((g a1) a2)... ak)`
- Espressioni funzionali possono costituire il corpo
  Dunque funzioni sono restituite da funzioni
- `val somma = fn x => (fn y => y+x);`
- `val tre = somma 1 2;`
- `val sommadue = somma 2;`
- `val cinque = sommadue 3;`

university-logo

Una notazione più compatta

- `fun f x = x*x;` è abbreviazione per
  `val f = fn x => x*x;`
- In generale: `fun F x1 x2 ... xn =` *`corpo`*`;`
  abbrevia
  `val F = fn x1=>(fn x2=>...(fn xn=>`*`corpo`*`)...);`
- Osserva: non funzione di n argomenti...
- Ma funzione che restituisce funzione!

university-logo

Nucleo del linguaggio

- Termini di base:
  - Valori di tipo base: $M ::= tt \mid ff \mid 0 \mid 1 \mid \cdots \mid n \mid \cdots$
  - Funzioni di tipo base: $M ::= and \mid \cdots \mid succ \mid + \mid \cdots$
  - Termini condizionali: $M, N, P ::= cond(M, N, P)$
- Termini propriamente funzionali:

$$M, N ::= x \mid (MN) \mid (\lambda x.M)$$

- Un meccanismo di ambiente per associare nomi a termini
  `let I =` M `in` N
- Abbreviazioni:
  `val I =` M, associa `I` a M in tutto quel che segue.
  `fun F x =` M, equivale a `val F =` $\lambda x.M$.

university-logo

Intermezzo: Tipi

- Linguaggio senza tipi. Dunque anche $+tt(\lambda x.x)$ è un termine
- Possiamo imporre dei tipi:
  - Tipi base: $T ::= bool \mid int, \ldots$
  - Tipi delle funzioni: $T, S ::= T \to S$
- I termini sono classificati secondo i tipi:

$$\vdash tt, ff : bool \qquad \Gamma, x : T \vdash x : T \ (Ax)$$

$$\vdash n : int \qquad \vdash + : int \times int \to int$$

$$\frac{\Gamma \vdash M : bool \quad \Gamma \vdash N : T \quad \Gamma \vdash P : T}{\Gamma \vdash cond(M, N, P) : T} \ (cond)$$

$$\frac{\Gamma, x : S \vdash M : T}{\Gamma \vdash \lambda x.M : S \to T} \ (\mathcal{I}\to) \qquad \frac{\Gamma \vdash M : S \to T \quad \Gamma \vdash N : S}{\Gamma \vdash MN : T} \ (\mathcal{E}\to)$$

university-logo

Redex e ridotti

- Un redex è un'espressione della forma:
  1. $f\ M_1 \cdots M_n$, dove f è una funzione primitiva di arità n e $M_1, \ldots, M_n$ sono tutte costanti di tipo base;
  2. $cond(v, N, P)$ dove v è tt oppure ff;
  3. $(\lambda x.M)N$ (questo è un β-*redex*).
- Il ridotto di un redex è l'espressione:
  1. Per un redex di tipo (1), la costante che rappresenta il valore di f applicata ai valori $M_1, \ldots, M_n$;
  2. Per un redex di tipo (2), il termine N se $v = tt$, altrimenti P;
  3. Per un β-redex, il termine $M[N/x]$, cioè la sostituzione (senza cattura) di N al posto delle occorrenze libere di x in M.
- La definizione presuppone quella di variabile libera e legata, sostituzione, ecc. (che non diamo)

university-logo

Computazioni

- Un identificatore legato in ambiente “sta per” il termine cui è legato;
- Un termine M si riduce in un passo in N ($M \rightarrow_1 N$) sse N si ottiene da M riscrivendo un redex presente in M con il suo ridotto
- $M \rightarrow N$ sse esiste una catena di riduzioni in un passo tale che $M \rightarrow_1 M_1 \rightarrow_1 M_2 \cdots \rightarrow_1 M_k \rightarrow_1 N$
  È ammesso $k = 0$: $M \rightarrow M$ (formalmente: $\rightarrow = \rightarrow_1^*$)
- M è in forma normale sse in M non sono presenti redex.
- Beta-conversione: La chiusura *simmetrica* della riduzione si dice conversione e si denota con $M \leftrightarrow N$ (o anche $M =_\beta N$).

university-logo

Termini con e senza forma normale

- $K = \lambda x.\lambda y.x$;
- $I = \lambda x.x$
- $\Delta = \lambda x.xx$;
- $\Delta\ \Delta$ non ha forma formale;
- $K\ I\ I$ ha forma normale I;
- $K\ I\ (\Delta\Delta)$ ha forma normale I, ma alcune riduzioni non terminano.

university-logo

Confluenza e forme normali

Theorem (Confluenza)

Se $M \twoheadrightarrow N_1$ e $M \twoheadrightarrow N_2$, allora esiste un termine P tale che $N_1 \twoheadrightarrow P$ e $N_2 \twoheadrightarrow P$.

Corollary

Se un termine ha una forma normale, questa è unica.

university-logo

Valori

- Dove terminare la riduzione di un termine?
- Prima risposta: sulla forma normale.
- Obiezione: che ne è di un termine come il seguente?

```
val G = λ x . ((λ y . y+1) 2);
```

- Dobbiamo valutare "dentro la dichiarazione"? In tal caso otterremmo

```
λ x . 3
```

- Ma nei linguaggi di programmazione *non* si valuta il corpo di una funzione fino alla sua chiamata!

university-logo

Valori: definizione

- I valori sono un sottinsieme dei termini
- Ogni costante di tipo base è un valore: $0, 1, n, tt, ff, \ldots$
- Una variabile è un valore
- Un'astrazione è un valore

Un programma verrà ridotto sino a raggiungere un valore.
Useremo L per indicare un generico valore.
Useremo a (atomo) per indicare una costante di tipo base o una variabile.

university-logo

Strategie di riduzione

- Un termine M può essere ridotto in molti modi.
- Un linguaggio di programmazione rimuove questo nondeterminismo, fissando una strategia di valutazione (un interprete) $Eval$:

$$Eval(M) \text{ è il valore, se esiste, del termine } M$$

- La strategia deve essere corretta rispetto alla β-riduzione:

$$Eval(M) = L \implies M \twoheadrightarrow L$$

- Quando vale anche l'inverso, la strategia è normalizzante.

university-logo

Strategia per valore, informale

1. Scandisci l'espressione da valutare a partire da sinistra, selezionando la prima applicazione che incontri (sia essa MN) oppure il primo $cond(M, N, P)$.
2. Se si tratta di $cond(M, N, P)$, valuta (applicando ricorsivamente questo stesso metodo) M, fino a ridurla ad un valore, che sarà tt oppure ff; il ridotto è allora, rispettivamente, N o P.
3. Se si tratta di un'applicazione, valuta M, fino a ridurla ad un valore; questo potrà essere:
   3.1 una funzione base f;
   3.2 un'astrazione $\lambda x.M$;
4. Valuta quindi la parte argomento N dell'applicazione, per ridurla ad un valore L.
5. Riduci infine il redex $(f\ L)$ oppure $(\lambda x.M')L$ e riparti da (1).

university-logo

Strategia per valore

$$Eval_V(a) = a \qquad Eval_V(\lambda x.M) = \lambda x.M$$

$$\frac{Eval_V(M) = tt}{Eval_V(cond(M,N,P)) = N}\ (c_{tt}) \qquad \frac{Eval_V(M) = ff}{Eval_V(cond(M,N,P)) = P}\ (c_{ff})$$

$$\frac{Eval_V(M) = f \quad Eval_V(N) = L \quad f(L) = a'}{Eval_V(MN) = a'}\ (f \text{ base})$$

$$\frac{Eval_V(M) = \lambda x.M' \quad Eval_V(N) = L}{Eval_V(MN) = M'[L/x]}\ (app)$$

university-logo

Considerazioni su $Eval_V$

- Non è normalizzante:
  $Eval_V(K\ I\ (\Delta\ \Delta))$ è indefinito.
  Tuttavia $K\ I\ (\Delta\ \Delta) \rightarrow I$.
- In presenza di "riuso" di un parametro formale, il valore viene calcolato una sola volta:
  $F = \lambda b.\lambda n.cond(b, n + n, n)$
  $F\ tt\ (3 + 3)$ ha valore 12;
  il redex $(3 + 3)$ viene valutato a 6 una sola volta, prima della chiamata di F.

Strategia per nome, informale

1. Scandisci l'espressione da valutare a partire da sinistra, selezionando la prima applicazione che incontri (sia essa MN) oppure il primo $cond(M, N, P)$.
2. Se si tratta di $cond(M, N, P)$, valuta (applicando ricorsivamente questo stesso metodo) M, fino a ridurla ad un valore, che sarà tt oppure ff; il ridotto è allora, rispettivamente, N o P.
3. Se si tratta di un'applicazione, valuta M, fino a ridurla ad un valore; questo potrà essere:
   3.1 una funzione base f;
   3.2 un'astrazione $\lambda x.M$;
4. Se si tratta di una funzione base f, valuta l'argomento N, e restituisci il valore $f(L)$.
5. Se si tratta di un'astrazione, valuta $M'[N/x]$ e riparti da (1).

Strategia per nome

$$Eval_V(a) = a \qquad Eval_V(\lambda x.M) = \lambda x.M$$

$$\frac{Eval_V(M) = tt}{Eval_V(cond(M, N, P)) = N}\ (c_{tt}) \qquad \frac{Eval_V(M) = ff}{Eval_V(cond(M, N, P)) = P}\ (c_{ff})$$

$$\frac{Eval_V(M) = f \quad Eval_V(N) = L \quad f(L) = a'}{Eval_V(MN) = a'}\ (f\ \text{base})$$

$$\frac{Eval_V(M) = \lambda x.M'}{Eval_V(MN) = M'[N/x]}\ (app)$$

university-logo

Considerazioni su $Eval_N$

- In presenza di "riuso" di un parametro formale, il valore viene calcolato tante volte quante sono le occorrenze del parametro formale:
  $F = \lambda b.\lambda n.cond(b, n + n, n)$
  $F\ tt\ (3 + 3)$ ha valore 12;
  il redex $(3 + 3)$ viene valutato a 6 due volte, come
  $(3 + 3) + (3 + 3)$, nel corpo di F.
- $Eval_N$ è una strategia normalizzante...

university-logo

$Eval_N$ è normalizzante

Theorem (di standardizzazione)

Se M riduce ad un valore, allora tale valore è calcolato dalla strategia per nome:

$$M \to L \implies Eval_N(M) = L$$

university-logo

Un confronto

- Per valore (applicativa):
  - Semplice implementazione
  - Efficiente
  - Non normalizzante
- Per nome (normale):
  - Implementazione più complessa
  - Inefficiente
  - Normalizzante
- Strategia lazy: prendere i vantaggi di entrambi

university-logo

Strategia lazy

- Implementazione della strategia normale
- Gli argomenti sono passati non valutati
- Quando avviene un riferimento al parametro formale, questo viene valutato
- Il valore viene salvato, così da poter essere riutilizzato per le occorrenze successive.
- Implementazione meno ovvia della strategia per valore
- Cruciale: la correttezza della lazy dipende dalla mancanza di effetti collaterali.

  La valutazione di una stessa espressione dà sempre lo stesso valore, indipendentemente dalla sua posizione

university-logo

Strategie by need

- Le strategie per nome e lazy sono esempi di strategie by need
- Una strategia è "by need" se un redex viene ridotto solo se necessario
- Più precisamente: un redex è necessario se compare (o se un suo residuo compare) in ogni riduzione normalizzante
- Le strategie by need tendono a ridurre il numero di redex distinti che sono ridotti
- Ma pagano ciò duplicando i redex (che dunque devono essere ridotti più volte)
- E comunque con le strutture dati più complesse che devono essere mantenute dalla macchina astratta

Linguaggi funzionali

- Arrischiscono il nucleo con tipi, costrutti, meccanismi
- Alcuni includono meccanismi imperativi
- In tal caso, possono verificarsi effetti collaterali
- Tutti i linguaggi con effetti collaterali adottano una strategia per valore
- Una veloce rassegna…

university-logo

Ambienti locali

- `let x =` *exp* `in` *exp1* `end`
- È zucchero sintattico per

  `(fn x =>` *exp1*`)` *exp*.

university-logo

Pattern matching, 1

- Ricorsione tediosa nella gestione dei casi terminali
- ```
  fun Fibo n = if n=0 then 1
               else if n=1 then 1
                    else Fibo(n-1)+Fibo(n-2);
  ```
- Con pattern matching:

  ```
  fun Fibo 0 = 1
   |  Fibo 1 = 1
   |  Fibo n = Fibo(n-1)+Fibo(n-2);
  ```
- Parametri formali non sono solo variabili, ma anche costanti e, più in generale, espressioni con costruttori (*pattern*).
- Alla chiamata il parametro attuale è confrontato con i pattern, in modo sequenziale, dall'alto al basso. Il primo pattern che "*matches*", seleziona il corpo (legando il parametro).

university-logo
  ```

Tipi di base strutturati: ennuple (*tuples*)

- Simili a record senza etichette
- `(4,5): int * int`
- Sono strutture eterogenee:

  `(3,"pippo",tt) : int * string * bool`
- La selezione si ottiene con pattern matching:

```
fun primo_di_tre (x,y,z) = x;
fun secondo_di_due (x,y) = y;
```

university-logo

Tipi di base strutturati: liste

- `[4,5]: int list`
- Una costante polimorfa: `nil: 'a list`
- Un costruttore per il cons:

  `3::[4,5] : int list`

  è un'espressione che denota la lista `[3,4,5]`
- `ff::nil : bool list`

  è la lista `[ff]`
- Sono strutture omogenee:

  `[3,"pippo",tt]` è sintatticamente sbagliata

  perché non rispetta i tipi
- La selezione si ottiene con pattern matching:

```
fun hd nil = errore
 |  hd x::l = x;
fun tl nil =errore
 |  tl x::l = l;
```

university-logo

Pattern matching, 2

- Con tipi strutturati il pattern matching dimostra tutta la sua flessibilità
- ```
  fun lung nil = 0
   |   lung e::resto = 1 + lung(resto);
  ```
- ```
  fun append nil  l = l
   |   append x::r l = x::(append r l);
  ```

university-logo

Polimorfismo

- Fa parte del nucleo del paradigma (se il linguaggio è tipizzato)
- Anche in assenza di indicazioni di tipo, la macchina astratta inferisce e controlla i tipi
- `lung : 'a list -> int`
- `hd : 'a list -> 'a`
  L'inferenza mette in relazione il tipo del risultato con quello dell'argomento
- `append 'a list -> 'a list -> 'a list`
  L'inferenza deduce che le due liste devono essere omogenee tra loro

university-logo

Ordine superiore

- Fa parte del nucleo del paradigma
- I costruttori di tipo possono essere applicati anche a tipi superiori

```
fun last nil = errore
 |  last x::nil = x
 |  last x::l = last l;
```

```
[last,hd]: ('a list -> 'a) list
(last,hd): ('a list -> 'a) list * ('a list -> 'a) list
```

university-logo

Funzioni di ordine superiore

- ```
  fun map f nil   = nil
   |  map f x::l = (f x) :: (map f l);

  map : ('a -> 'b) -> ('a list) -> ('b list)
  ```
- ```
  fun fold f i nil   = i
   |  fold f i x::l = f (x, fold f i l);

  fold : ('a*'a -> 'b) -> 'a -> ('a list) -> 'b
  ```
- Possono essere usate come schemi per la generazione di programmi

university-logo

Tipi definiti dall'utente

- Definizione di arbitrari tipi (algebre) mediante costanti e costruttori.
- Usiamo la notazione di ML.
- ```
  - datatype Expense =
      FOOD of int
    | HOUSE of int
    | BILLS of int
    ;
  > datatype Expense
    con FOOD = fn : int -> Expense
    con HOUSE = fn : int -> Expense
    con BILLS = fn : int -> Expense
  -
  ```
- ```
  val thisWeek =
     [FOOD 30, HOUSE 5, FOOD 4, FOOD 6, BILLS 10];
  ```

university-logo

Tipi induttivi

- ```
  - datatype NatNum = Zero | Succ of NumNat;
  > datatype NatNum
    con Zero = Zero : NatNum
    con Succ = fn : NumNat -> NumNat
  ```
- ```
  - datatype IntTree =
      NIL
    | NODE of int * IntTree * IntTree
    ;
  > datatype IntTree
    con NIL = NIL : IntTree
    con NODE = fn : int * IntTree * IntTree -> IntTree
  ```
- Selettori definibili con pattern-matching.

university-logo

Tipi (induttivi) polimorfi

- ```
  - datatype 'a Tree =
      NIL
    | NODE of 'a * ('a Tree) * ('a Tree)
    ;
  > datatype 'a Tree
    con NIL = NIL : 'a Tree
    con NODE = fn : 'a * 'a Tree * 'a Tree -> 'a Tree
  ```

- ```
  - fun preOrder NIL = [ ]
    |   preOrder NODE (i,left,right) =
            (preOrder left) @ (preOrder right) @ [i]
    ;
  > val preOrder = fn : 'a Tree -> 'a list
  ```

  `@` è *append*, infisso

university-logo

Aspetti imperativi

- Per ogni tipo `T` è definito il tipo `T ref` delle variabili modificabili che contengono valori di tipo `T`.
- Il costrutto

  `ref` v

  crea una variabile modificabile di tipo `T`, inizializzata al valore v (di tipo `T`)
- `val I = ref 4;`
- Dereferenziazione esplicita, con operatore `!`

  ```
  val n = !I + 1;
  ```

- Modifica con assegnamento

  ```
  I := !I + 1;
  ```

university-logo

Altri effetti collaterali

- Altri effetti collaterali:
  - I/O
  - eccezioni (o terminazioni anticipate)
  - event programming
- Come gestire questi casi in un linguaggio funzionale puro?
- In particolare, come gestirli in un linguaggio lazy?
- In tali linguaggi l'assenza di effetti collaterali è cruciale per la correttezza dell'implementazione!
- Vedremo nel seguito la tecnica delle monadi.

university-logo

Valori di tipo strutturato

- In linguaggio per valore, una valore di tipo `T list` è una lista di valori di tipo `T`.
- In un linguaggio lazy, non sarebbe una buona definizione perché violerebbe la filosofia by need
  - `hd [2, ((fn n=>n+1) 2)];`

    il redex dentro la lista non è necessario
  - Sarà valutato solo quando *quel* valore è necessario

    `hd (tl [2, ((fn n=>n+1) 2)]);`
- In linguaggio lazy, i valori di tipo strutturato sono arbitrarie espressioni di quel tipo (anche con redex)
  - Analogo al trattamento delle λ-astrazioni: non si riduce dentro un'astrazione
  - Non si riduce "dentro" una lista, o una tupla

university-logo

Oggetti infiniti

- In linguaggi con strategia per nome (o lazy), possiamo manipolare oggetti potenzialmente infiniti
- `val infiniti2 = 2 :: infiniti2;`

  È una lista "infinita" di 2.
- Possiamo manipolarla:

```
hd infiniti2;
hd (tl (tl (tl infiniti2)));

fun numeriDan n = n :: numeriDan(n+1);
fun quadratiDan2 n = map (fn y => y*y) (numeriDan n);
```

university-logo

Una macchina astratta: SECD

- Peter Landin, 1964
- Introduce il concetto di *chiusura* (generalizza i *thunk* di Algol).
- Quattro componenti, organizzate come *pile*:
  Stack, Environment, Control, Dump.
  - Controllo: il programma da eseguire, in sintassi astratta
  - Stack: i valori temporanei (delle sottoespressioni) valutati sin qui
  - Ambiente: legami identificatori-valori
  - Dump: memorizzazione della computazione corrente (tutto lo stato (S, E, C, D)), al momento in cui si esegua una chiusura (nel suo ambiente).
- Notazione: $cl(E, M, x)$ chiusura di M, con var legata x, nell'ambiente E.

university-logo

Controllate da cosa è presente sul Controllo

- $[S, E, c :: C, D] \longrightarrow [c :: S, E, C, D]$
- $[S, E, x :: C, D] \longrightarrow [E(x) :: S, E, C, D]$
- $[S, E, (MN) :: C, D] \longrightarrow [S, E, M :: N :: @ :: C, D]$
- $[S, E, (\lambda x.M) :: C, D] \longrightarrow [cl(E, M, x) :: S, E, C, D]$
- $[v :: f :: S, E, @ :: C, D] \longrightarrow [f(v) :: S, E, C, D]$
- $[v :: cl(E, M, x) :: S, E, @ :: C, D]$
  $\longrightarrow [\square, E_1[x \leftarrow v], [M], (S, E, C) :: D]$
- $[v :: S, E, \square, (S', E', C,) :: D] \longrightarrow [v :: S', E', C', D]$
- $[S, E, \square, \square]$ stop

Inizio: $[\square, \square, M, \square]$

university-logo

Esempio

Valutiamo $(F\ 3)$, dove

$$F = \lambda x.(sqrt\ ((\lambda y.succ\ x)\ x)).$$

S	**E**	**C**	**D**	**reg**
$\square$	$\square$	$[(F\ 3)]$	$\square$	
$\square$	$\square$	$[F, 3, @]$	$\square$	3
$[ch_1]$	$\square$	$[3, @]$	$\square$	4
	dove $ch_1 = cl(\square, (sqrt\ ((\lambda y.succ\ x)\ x)), x)$			
$[3, ch_1]$	$\square$	$[@]$	$\square$	1
$\square$	$[x \leftarrow 3]$	$[(sqrt\ ((\lambda y.succ\ x)\ x))]$	d_1	5
	dove $d_1 = (\square, \square, \square) :: \square$			
$\square$	$[x \leftarrow 3]$	$[sqrt, ((\lambda y.succ\ x)\ x), @]$	d_1	3
$[sqrt]$	$[x \leftarrow 3]$	$[((\lambda y.succ\ x)\ x), @]$	d_1	1
$[sqrt]$	$[x \leftarrow 3]$	$[((\lambda y.succ\ x)\ x), x, @, @]$	d_1	1

university-logo

Esempio, segue

$[ch_2, sqrt]$	$[x \leftarrow 3]$	$[x, @, @]$	d_1	4
	dove $ch_2 = cl([x \leftarrow 3], (succ\ x), y)$			
$[3, ch_2, sqrt]$	$[x \leftarrow 3]$	$[@, @]$	d_1	2
$\square$	$[x \leftarrow 3, y \leftarrow 3]$	$[(succ\ x)]$	d_2	5
	dove $d_2 = ([sqrt], [x \leftarrow 3], [@]) :: d_1$			
$\square$	$[x \leftarrow 3, y \leftarrow 3]$	$[succ, x, @]$	d_2	3
$[succ]$	$[x \leftarrow 3, y \leftarrow 3]$	$[x, @]$	d_2	1
$[3, succ]$	$[x \leftarrow 3, y \leftarrow 3]$	$[@]$	d_2	2
$[4]$	$[x \leftarrow 3, y \leftarrow 3]$	$\square$	d_2	5
$[4, sqrt]$	$[x \leftarrow 3]$	$[@]$	d_1	6
$[2]$	$[x \leftarrow 3]$	$\square$	d_1	5
$[2]$	$\square$	$\square$	$\square$	6