Lambda Calcolo

Luca Abeni

May 16, 2017

Lambda? (λ!)

- Definizione di funzione: sintassi differente rispetto ad applicazione
  - In ML, **fn** x => e
  - `x`: identificatore per il parametro formale
  - `e`: espressione (sperabilmente usa `x`)
- Sostituendo `fn` con λ e il simbolo => con .
  - $\lambda x.e$
  - x: variabile legata
  - e: espressione
- Abbiamo appena scoperto il lambda-calcolo!!!
  - Ma... A che serve?
  - Formalismo matematico per quanto detto fin'ora... Definizioni rigorose!

Definizione Formale

- Espressione nel lambda-calcolo: nome, funzione o applicazione di espressioni a espressioni
  - funzione: λnome.espressione
  - applicazione: espressione espressione
- Più formalmente, `e = x | λx.e | e e`
  - `x` indica un identificatore (variabile, costante...)
  - `e` indica una generica espressione
- Dal punto di vista pratico, aggiungiamo parentesi per rendere più leggibile!!!
  - `e = x | (λx.e) | (e e)`
  - Non strettamente necessario, ma $(((f_1 f_2) f_3) f_4)$ è più chiaro di $f_1 f_2 f_3 f_4$...

Lambda-Calcolo vs Paradigma Funzionale

- Epressioni del lambda-calcolo: si identificano subito astrazione ($\lambda x.e$) ed applicazione ($e\ e$)
  - Astrazione: lega variabile x nell'espressione e
    - Se si cambia λx in λy e si cambiano tutte le x di e in y, la semantica di e non cambia!!!
    - Le sostituzioni dovute ad applicazione non hanno effetto su x
  - Applicazione: richiede sostituzione (senza cattura!!!)

Lambda-Calcolo vs Paradigma Funzionale - 2

- Lambda-calcolo: astrazione ed applicazione
- Concetti utilizzabili per la computazione di programmi funzionali
- Lambda-calcolo permette di formalizzare le cose, basandosi s definizioni matematicamente solide
  - Partiamo da variabili libere e legate

Variabili Libere e Variabili Legate

- Informalmente: una variabile x è *legata* da $\lambda x.$; una variabile è libera se non è legata da nessun λ
- Formalmente: $F_v(e)$: variabili libere nell'espressione e; $B_v(e)$: variabili legate nell'espressione e
  - Se x è un identificatore, $F_v(x) = \{x\}$ e $B_v(x) = \emptyset$
    - Se un'espressione è composta solo da una variabile, quella variabile è libera
  - $F_v(e_1 e_2) = F_v(e_1) \cup F_v(e_2)$ e $B_v(e_1 e_2) = B_v(e_1) \cup B_v(e_2)$
    - Componendo due espressioni, non si "cambia lo stato" (libere o non libere) delle loro variabili

Variabili Legate - Ancora

- Definizione formale di variabile legata (continua):
  - $F_v(\lambda x.e) = F_v(e) \setminus \{x\}$ e $B_v(\lambda x.e) = B_v(e) \cup \{x\}$
    - L'operatore λ (astrazione) rimuove una variabile dalle variabili libere e la aggiunge alle variabili legate
- Semplice, no?

Sostituzione

- Definizione formale di sostituzione!
  - Basata su $F_v(e)$ e $B_v(e)$
  - $e[e'/x]$: nell'espressione e, sostituisci x con e'
  - Risultato della sostituzione indicato con $\rightarrow$
- Definizione per tre casi: valore, astrazione ed applicazione:
  - Se x è un identificatore, $x[e'/x] = e'$
  - Se $x \neq y$, $y[e'/x] = y$
    - Sostituendo x con e' nell'espressione "x", si ottiene e'
    - Sostituendo x con e' nell'espressione "y", non cambia

Sostituzione - 2

- Sostituzione in caso di applicazione:
  - $(e_1 e_2)[e'/x] = (e_1[e'/x] e_2[e'/x])$
- Sostituzione in caso di astrazione:
  - Se $x \neq y$ e $y \notin F_v(e')$, $(\lambda y.e)[e'/x] = (\lambda y.e[e'/x])$
    - $y \notin F_v(e')$: metodo semplice per evitare cattura di y!!!
  - Se $x = y$, $(\lambda y.e)[z/x] = (\lambda y.e)$
    - Sostituire la variabile legata da λ non cambia nulla...

Sostituzione - Cattura

- Se $x \neq y$ e $y \notin F_v(e')$, $(\lambda y.e)[e'/x] = (\lambda y.e[e'/x])$
  - $y \notin F_v(e')$: metodo semplice per evitare cattura di y!!!
  - Cosa succede se $y \in F_v(e')$?
- Per evitare cattura, si rinomina la variabile legata da λ!
  - Una funzione non deve dipendere dal nome del parametro formale...
  - $\lambda x.x = \lambda y.y$ e simili... (in generale: $\lambda x.e = \lambda y.(e[y/x])$
- Possibile rinominare in modo da usare una variabile non libera in e'!

Esempio di Cattura

- Consideriamo l'espressione $\lambda x.xy$ e proviamo a sostituire y con xz
  - $(\lambda x.xy)[xz/y]$
- Applicando $(\lambda y.e)[e'/x] \rightarrow \lambda y.(e[e'/x])$ senza fare attenzione si ottiene
  - $(\lambda x.xy)[xz/y] \rightarrow \lambda x.(xy[xz/y]) = \lambda x.xxz$
  - Abbiamo "catturato" la x di xz...
  - Chiaro il problema?
- Soluzione: cambiamo $\lambda x.xy$ in $\lambda v.vy$
  - $(\lambda v.vy)[xz/y] \rightarrow \lambda v.(vy[xz/y]) = \lambda v.vxz$
  - Ora va meglio...

Equivalenza fra Espressioni

- Date due espressioni e_1 ed e_2, quando si possono dichiarare equivalenti?
  - Risposta intuitiva: quando differiscono solo per il nome di variabili legate!
- Data una variabile y non presente in e, $\lambda x.e \equiv \lambda y.e[y/x]$
  - Cambia λx in λy
  - Cambia tutte le occorrenze di x in y dentro all'espressione e
- Alfa Equivalenza!!! $\equiv_\alpha$
- Due espressioni sono α-equivalenti anche se una si ottiene dall'altra sostituendo una sua parte con una parte α-equivalente

Dopo α, ... β!

- Come noto, computazione per riscrittura / semplificazione...
- Più formalmente, β-riduzione!!!
  - $(\lambda x.e)e' \rightarrow_\beta e[e'/x]$
- e_1 è β-ridotta ad e_2 quando e_2 si ottiene da e_1 tramite β-riduzione di qualche sotto espressione
  - Nota: $(\lambda x.e)e'$ è un redex!
  - E $e[e'/x]$ è il suo ridotto...
  - Come si procede se ci sono più redex? Non importa! (confluenza)

β Riduzione

- β riduzione: introduce una relazione fra espressioni
- Relazione non simmetrica: $e_1 \rightarrow_\beta e_2 \not\Rightarrow e_2 \rightarrow_\beta e_1$
  - Quindi, non è una relazione di equivalenza...
  - Esiste però β-equivalenza $=_\beta$ (chiusura riflessiva e transitiva di $\rightarrow_\beta$)
- Informalmente: $e_1 =_\beta e_2$ significa che esiste una catena di β-riduzioni che "collega" e_1 ed e_2
  - La "direzione" delle β-riduzioni non conta!

β Equivalenza

- β-equivalenza $=_\beta$: definita da β-riduzione $\rightarrow_\beta$
  - Chiusura riflessiva e transitiva di $\rightarrow_\beta$...
  - Ma che significa???
- Prendiamo la relazione $e_1 \rightarrow_\beta e_2$ ed estendiamola per essere riflessiva ($e_1 =_\beta e_2 \Rightarrow e_2 =_\beta e_1$) e transitiva ($e_1 =_\beta e_2 =_\beta e_3 \Rightarrow e_1 =_\beta e_3$)
  - $e_1 \rightarrow_\beta e_2 \Rightarrow e_1 =_\beta e_2$
  - $\forall e, e =_\beta e$
  - $e_1 =_\beta e_2 \Rightarrow e_2 =_\beta e_1$
  - $e_1 =_\beta e_2 =_\beta e_3 \Rightarrow e_1 =_\beta e_3$

Forme Normali

- Espressione che non contiene redex $\rightarrow$ no β-riduzioni
  - Forma normale
  - $\lambda x.\lambda y.x$ è in forma normale, $\lambda x.(\lambda y.y)x$ no $((\lambda y.y)x \rightarrow_\beta x$, quindi $\lambda x.(\lambda y.y)x =_\beta \lambda x.x)$
- β-riduzione può terminare in una forma normale...
- ...O può procedere all'infinito!
  - $(\lambda x.xx)(\lambda x.xx) \rightarrow_\beta (xx)[(\lambda x.xx)/x] = (\lambda x.xx)(\lambda x.xx)$...
- Come ricorsione infinita o loop infiniti...

Confluenza

- Torniamo a β-riduzioni di espressioni con più redex...

  "Se e si riduce a e_1 con qualche passo di (β-)riduzione ed e si riduce a e_2 con qualche passo di riduzione, allora esiste e_3 tale che sia e_1 che e_2 si riducono a e_3 con qualche passo di riduzione"

- Se e è riducibile in forma normale, allora questa non dipende dall'ordine in cui si sono ridotti i redex.

Espressività del λ-Calcolo

- Il λ calcolo come definito fino ad ora può apparire "limitante"
  - Espressioni composte solo da variabili, applicazione ed astrazione...
  - $\lambda x.x + 2$ non è un'espressione valida
    - 2 non è una variabile; $+$ non è una variabile!
- Il λ calcolo ha però una grande potenza espressiva...
  - Turing equivalente: può codificare tutti gli algoritmi "sensati"
  - Permette allora di codificare costanti, operazioni aritmetiche, etc...
    - Come?

Esempio: Codifica dei Naturali

- Definizione basata su assiomi di Peano:
  - 0 è un naturale
  - Se n è un naturale, il successivo di n (succ(n)) è anch'esso un naturale
- Church ha fatto qualcosa di simile...
  - 0 è codificato come $\lambda f.\lambda x.x$ (funzione f applicata 0 volte a x)
  - succ(n): applica f ad n
- In pratica: 0 = funzione appilcata 0 volte ad una variabile, 1 = funziona applicata 1 volta, ...
- n: funzione applicata n volte ad una variabile
- Ma come si definisce formalmente il successivo di un numero?

Naturali: Successivo

- succ(n) = $\lambda n.\lambda f.\lambda x.f((nf)x)$
  - Aggiungi una f a n???
- Informalmente: n rappresentato da $\lambda f.\lambda x.$ seguito da n volte f e da x
  - "Corpo" della funzione n: $\overbrace{f(\ldots f(}^{n}\, x)\ldots)$
  - Va "estratto" da n (rimuovendo $\lambda f.\lambda x.$), aggiunta una f e astratto di nuovo rispetto a f e x
- Vediamo come fare:
  - "Estrazione": si applica n ad f ed $x \rightarrow ((nf)x)$
  - Aggiunta di f: facile... $\rightarrow f((nf)x)$
  - Ri-astrazione: $\lambda f.\lambda x.f((nf)x)$
- Il tutto è funzione di n: $\lambda n.\lambda f.\lambda x.f((nf)x)$

Codifica dei Naturali - 1, 2, ...

- $1 = \text{succ}(0)$: $(\lambda n.\lambda f.\lambda x.f((nf)x))(\lambda f.\lambda x.x)$
  - $(\lambda n.\lambda g.\lambda y.g((ng)y))(\lambda f.\lambda x.x)$
  - $\lambda g.\lambda y.g(((\lambda f.\lambda x.x)g)y)$
  - $\lambda g.\lambda y.g((\lambda x.x)y) = \lambda g.\lambda y.gy$
  - $\lambda g.\lambda y.gy = \lambda f.\lambda x.fx$
- $2 = \text{succ}(1)$: $(\lambda n.\lambda f.\lambda x.f((nf)x))(\lambda f.\lambda x.fx)$
  - $(\lambda n.\lambda g.\lambda y.g((ng)y))(\lambda f.\lambda x.fx)$
  - $\lambda g.\lambda y.g(((\lambda f.\lambda x.fx)g)y)$
  - $\lambda g.\lambda y.g((\lambda x.gx)y)$
  - $\lambda g.\lambda y.g(gy) = \lambda f.\lambda x.f(fx)$
- Analogamente, $3 = \text{succ}(2) = \lambda f.\lambda x.f(f(fx))$, etc...

Naturali: Somma

- Come detto, $n \equiv f$ applicata n volte ad x
- Quindi, $2 + 3 =$ “applica 2 volte f a 3” (3 volte f a 2...)
  - Applica 2 volte f ad “applica 3 volte f ad x”...
- $n + m$: applica n volte f a m
  - Estrai il corpo di n e di m
  - Nel corpo di n, sostituisci x con m
  - Ri-astrai il tutto rispetto ad f e ad x
  - Astrai rispetto ad m ed n
- Come fare:
  - Corpo di m: $(mf)x$
  - Corpo di n con il corpo di m al posto di x: $(nf)((mf)x)$
  - Quindi, $\lambda n.\lambda m.\lambda f.\lambda x.(nf)((mf)x)$

Somma: Esempio

- $2 + 3$: $\lambda f.\lambda x.f(fx)$ + $\lambda f.\lambda x.f(f(fx))$
  - +: $\lambda n.\lambda m.\lambda f.\lambda x.(nf)((mf)x)$
- $(\lambda n.\lambda m.\lambda f.\lambda x.(nf)((mf)x))(\lambda f.\lambda x.f(fx))(\lambda f.\lambda x.f(f(fx)))$
  - $(\lambda n.\lambda m.\lambda g.\lambda y.(ng)((mg)y))(\lambda h.\lambda z.h(hz))(\lambda f.\lambda x.f(f(fx)))$
  - $\lambda g.\lambda y.((\lambda h.\lambda z.h(hz))g)(((\lambda f.\lambda x.f(f(fx)))g)y)$
  - $\lambda g.\lambda y.(\lambda z.g(gz))((\lambda x.g(g(gx)))y)$
  - $\lambda g.\lambda y.(\lambda z.g(gz))(g(g(gy)))$
  - $\lambda g.\lambda y.(g(g(g(g(gy)))))$
- Che è uguale a $\lambda f.\lambda x.f(f(f(f(fx))))$
  - f applicata 5 volte ad x: 5!
  - Infatti, $2 + 3 = 5$...

Si Può Fare...

- Analogamente a naturali ed operazioni aritmetiche, si possono implementare molte altre cose...
  - Tutto quel che serve!
  - Booleani, condizioni (`if ... then ... else`), ...
- Ma questo non vuol dire che la notazione sia semplice!
  - $2 + 3 \equiv$
    $(\lambda n.\lambda m.\lambda f.\lambda x.(nf)((mf)x))(\lambda f.\lambda x.f(fx))(\lambda f.\lambda x.f(f(fx)))$
- $\lambda x.x + 2$ non è un'espressione valida...
  - Ma $\lambda x.((\lambda n.\lambda m.\lambda f.\lambda x.(nf)((mf)x))x(\lambda f.\lambda x.f(fx))$ lo è!
  - E vuole dire la stessa cosa...

Estensione a λ Calcolo

- Lieve abuso di notazione: permettiamo di usare numeri naturali, operazioni, condizioni e quant'altro serve nelle espressioni
  - Tanto si può implementare tutto usando solo variabili, applicazione ed astrazione...
- Espressioni come $\lambda x.(x + 2)$ o $\lambda x.$`if x = 1 then 0 else ...` diventano valide!
  - Se vogliamo fare i puristi, possiamo sempre sostituire 2, $+$, `if ...` con la loro codifica...
- Versione "estesa" del λ calcolo puro (che ha solo variabili, astrazione ed applicazione)

Iterazione / Ricorsione

- Come implementare iterazione nel λ calcolo?
  - Paradigma funzionale: tramite ricorsione!
  - Allora: come implementare ricorsione???
- Occorrerebbe “dare un nome” a λx....
  - Ma questo non è possibile! No nozione di “ambiente esterno”...
- Implementiamo ricorsione usando solo astrazione ed applicazione...
- Esempio stupido:
  val rec f = **fn** n => **if** n = 0 **then** 0 **else** 1 + f (n − 1)
  - Parecchio stupida, ma è un esempio...
  - E’ abbastanza ovvio che implementa l’identità
    val f = **fn** n => n

λ Calcolo e Ricorsione - Esempio

- $f = \lambda n.\texttt{if n = 0 then 0 else 1 + } f\texttt{ (n - 1)}$
- "$f =$" non come definizione, ma come equazione...
  - $f = G(f)$... $G()$ funzione di ordine superiore
    - Riceve una funzione come argomento
    - Ritorna una fuzione come risultato
  - Risolvendo, si trova f... Ma, cosa significa "$=$"?
- Come risolvere questa equazione?
- Prima di tutto, definiamo G astraendo rispetto ad f:
- $G = \lambda f.\lambda n.\texttt{if } n = 0 \texttt{ then 0 else 1 + } f\texttt{ (n-1)}$
- Quindi, si tratta di trovare $h : h =_\beta Gh$
  - Applicando G ad h si ottiene ancora h (con β-uguaglianza!)

Ricorsione - Continua Esempio

- Da $f = \lambda n.\texttt{if}\ n = 0\ \texttt{then}\ 0\ \texttt{else}\ 1\ +\ f\,\texttt{(n-1)}$ a $\lambda f.\lambda n.\texttt{if}\ n = 0\ \texttt{then}\ 0\ \texttt{else}\ 1\ +\ f\,\texttt{(n-1)}$
  - Uh? Abbiamo Eliminato la Ricorsione!!!
  - Funzione da invocare ricorsivamente passata come parametro!
- Esempio:

```
val rec f = fn n => if n=0 then 0 else 1 + f(n-1)
```

$\Rightarrow$

```
val g = fn f => fn n => if n=0 then 0 else 1+f(n-1)
```

- Cerchiamo `f1` tale che `f1 = g f1`...
- Notare la scomparsa di `rec`

λ, α, β, ... Y???

- Torniamo al nostro problema: data una funzione G trovare $f : f =_\beta Gf$
  - Qui, "$=$" dopo qualche β-riduzione a destra o sinistra... β-uguaglianza!
- Equivale a trovare il *punto fisso* (fixpoint) di G...
- Come si fa? Y combinator!
  $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$
  - Uh??? E come funziona??? Consideriamo espressione e e calcoliamo Ye...

Y!!!

- $Ye = (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))e$
- $(\lambda x.e(xx))(\lambda x.e(xx)) = (\lambda y.e(yy))(\lambda x.e(xx))$
- $e(\lambda x.e(xx))(\lambda x.e(xx))$
- Ma $(\lambda x.e(xx))(\lambda x.e(xx))$ può essere il risultato di una β-riduzione...
  - $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ applicato ad e
- $e(\lambda x.e(xx))(\lambda x.e(xx)) =_\beta$
  $e(\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))e) =_\beta e(Ye)$
  - Nota: passaggi non per β-riduzione!
- $Ye = e(Ye) \Rightarrow YG = G(YG)$: YG è un punto fisso per G!!!

Y... Combinator???

- Y Combinator: $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$
- Combinator: espressione λ senza variabili libere
  - $\lambda f.$ ...
  - Funzione di ordine superiore: ha una funzione (G) come argomento e genera una funzione come risultato
  - Non contiene variabili libere! Tutti i simboli sono legati da qualche λ
- Come si vede Y è un'espressione $\lambda f.$ ... e non contiene variabili libere $\rightarrow$ è un combinator!
- E' un particolare combinator, che data una funzione ne calcola il punto fisso (fixed point combinator)
  - Non è l'unico... Ne esistono infiniti altri!
  - Funziona usando β-uguaglianza

Fixed Poing Combinators

- Importanza: permettono di implementare ricorsione in λ-calcolo
  - In ML, permettono ricorsione senza `val rec`!
  - WTH???
- Y Combinator: funziona con valutazione per nome
  - Con valutazione per valore, ricorsione infinita...
- Altri fixed point combinator funzionano con valutazione per valore
  - Z Combinator: $\lambda f.((\lambda x.(f(\lambda y.(xx)y)))(\lambda x.(f(\lambda y.(xx)y))))$
  - H Combinator: $\lambda f.((\lambda x.xx)(\lambda x.(f(\lambda y.(xx)y))))$

Fixed Poing Combinators in ML

- Implementazione in ML: possibile, ma... Non proprio facile!
- Problema con valutazione per valore di ML...
  - Non si implementa Y, ma un altro fixed point combinator
- Problema coi tipi delle funzioni...
  - Vanno usati tipi di dati ricorsivi per eliminare ricorsione dalle funzioni!
- Per gli interessati, vedere dispense!