Il λ-calcolo e la teoria del calcolo

Il Lambda-calcolo è il modello computazionale su cui si basano i linguaggi funzionali.

È un formalismo matematico molto semplice. Nella teoria del λ-calcolo si è in grado di formalizzare, studiare e indagare proprietà e nozioni comuni a tutti i linguaggi funzionali, senza essere gravati da tutti i tecnicismi dei linguaggi attuali, inutili dal punto di vista teorico.

In un certo senso, il λ-calcolo è un linguaggio funzionale paradigmatico ed estremamente semplice.

I concetti su cui si basa il calcolo Lambda sono quelli fondamentali in tutti i linguaggi di programmazione funzionale:

variabile ( formalizzabile con x, y ,z,…)

astrazione (funzione anonima) (formalizzabile da λx.M dove M è un termine e x una variabile)

applicazione (formalizzabile da MN, dove M e N sono termini )

Abbiamo visto che queste sono davvero nozioni fondamentali nella programmazione funzionale. Sembra però che vi siano altre nozioni importanti: elementi di base, operatori di base e possibilità di dare nomi alle espressioni.

Anche se può sembrare molto strano, queste nozioni in realtà non sono proprio fondamentali: possono essere derivate dalle prime.

Ciò significa che il nostro modello computazionale funzionale può essere basato solo sui concetti di variabile, astrazione funzionale e applicazione.

Usando la formalizzazione di questi tre concetti formiamo i termini lambda. Questi termini rappresentano sia i programmi che i dati nel nostro modello computazionale (la cui forza infatti dipende fortemente dal fatto che non distingue tra "programmi" e "dati".)

La definizione formale dei termini del Lambda-calculus (lambda-terms) è data dalla seguente grammatica:

Λ ::= X | (ΛΛ) | λX.Λ

dove Λ sta per l'insieme dei termini lambda e X è una metavariabile che si estende sull'insieme (numerabile) di variabili (x, y, v, w, z, x2, x2,...)

Variabile e applicazione sono concetti ben noti, ma cos'è l'astrazione?

L'astrazione è necessaria per creare funzioni anonime (cioè funzioni senza nome). Abbiamo visto che poter specificare una funzione anonima è molto importante nella programmazione funzionale, ma è anche importante poter associare un identificatore a una funzione anonima. Tuttavia possiamo evitare di introdurre nel nostro modello la nozione di "dare un nome a una funzione". Ci si potrebbe chiedere: come è possibile definire una funzione come fac senza poterla riferire tramite il suo nome??? Credimi, è possibile!.

Il termine λx. M rappresenta la funzione anonima che, dato un input x, restituisce il "valore" del corpo M.

Altre nozioni, generalmente considerate primitive, come gli operatori di base (+, \*, -, ...) o i numeri naturali, non sono strettamente necessarie nel nostro modello computazionale (anzi possono effettivamente essere trattati come concetti derivati).

Convenzioni di bracketing

Per motivi di leggibilità, è utile abbreviare le astrazioni e le applicazioni nidificate come segue:

(λx1.(λx2. ... . (λxn.M) ... ) è abbreviato con (λx1x2 ... xn.M)

( ... ((M1M2)M3) ... Mn) è abbreviato con (M1M2M3 ... Mn)

Usiamo anche la convenzione di eliminare le parentesi più esterne e quelle che racchiudono il corpo di un'astrazione.

Esempio:

(λx.(x (λy.(yx)))) può essere scritto come λx.x(λy.yx)

In un'astrazione λx.P, si dice che il termine P è l'ambito dell'astrazione.

In un termine, l'occorrenza di una variabile si dice vincolata se si verifica nell'ambito di un'astrazione (cioè rappresenta l'argomento nel corpo di una funzione).

Si dice che sia libero altrimenti.

Questi due concetti possono essere formalizzati dalle seguenti definizioni.

Definizione di variabile vincolata

Definiamo BV(M), l'insieme delle Variabili Limitate di M, per induzione sul termine come segue:

BV(x) = Φ (l'insieme vuoto)

VV(PQ) = VV(P) U VV(Q)

BV(λx.P) = {x} U BV(P)

Definizione di variabile libera

Definiamo FV(M), l'insieme delle Variabili Libere di M, per induzione sul termine come segue:

FV(x) = {x}

FV(PQ) = FV(P) U FV(Q)

FV(λx.P) = FV(P) \ {x}

Sostituzione

La notazione M [L / x] denota il termine M in cui qualsiasi occorrenza libera (cioè che non rappresenta alcun argomento di una funzione) della variabile x in M ​​è sostituita dal termine L.

Definizione di sostituzione (per induzione sulla struttura del termine)

1. Se M è una variabile (M = y), allora:

y [L/x] ≡

{

L se x=y

y se x≠y

2. Se M è un'applicazione (M = PQ), allora:

PQ [L/x] = P[L/x] Q[L/x]

3. Se M è un'astrazione lambda (M = λy.P), allora:

λy.P[L/x] ≡

{

λy.P se x=y

λy.(P [L /x]) se x≠y

Si noti che in un'astrazione lambda viene eseguita una sostituzione solo se la variabile da sostituire è libera.

È necessaria molta attenzione quando si esegue una sostituzione, come mostra il seguente esempio:

Esempio:

Entrambi i termini (λx.z) e (λy.z) rappresentano ovviamente la funzione costante che restituisce z per ogni argomento

Supponiamo di voler applicare ad entrambi questi termini la sostituzione [x / z] (sostituiamo x con la variabile libera z). Se non teniamo conto del fatto che la variabile x è libera nel termine x e vincolata in (λx.z) otteniamo:

(λx.z) [x / z] => λx.(z[x / z]) => λx.x che rappresenta la funzione di identità

(λy.z) [x / z] => λy.(z[x/ z]) => λy.x che rappresenta la funzione che restituisce sempre x

Questo è assurdo, poiché entrambi i termini iniziali intendono denotare la stessa cosa (una funzione costante che restituisce z), ma applicando la sostituzione otteniamo due termini che denotano cose molto diverse.

Quindi una condizione necessaria per eseguire correttamente la sostituzione M[L/x] è che le variabili libere in L non cambino stato (cioè diventino non libere) dopo la sostituzione. Più formalmente: FV(L) e BV(M) devono essere insiemi disgiunti.

Tale problema è presente in tutti i linguaggi di programmazione, anche imperativi.

Esempio:

Consideriamo la seguente sostituzione:

(λz.x) [zw/x]

La variabile libera z nel termine da sostituire diverrebbe vincolata dopo la sostituzione. Questo non ha senso. Quindi, quello che deve essere fatto è rinominare le variabili legate nel termine su cui viene eseguita la sostituzione, affinché sia ​​soddisfatta la condizione sopra indicata:

(λq.x) [zw/x]

ora, effettuando la sostituzione, otteniamo correttamente: (λq.zw)

L'esempio sopra introduce e giustifica la nozione di convertibilità alfa

La nozione di alfa-convertibilità

Una variabile associata viene utilizzata per rappresentare dove, nel corpo di una funzione, viene utilizzato l'argomento della funzione. Per avere un calcolo significativo, i termini che differiscono solo per i nomi delle loro variabili legate devono essere considerati identici. Potremmo introdurre tale nozione di identità in modo molto formale, definendo la relazione di α-convertibilità. Ecco un esempio di due termini in tale relazione.

λz.z =αλx.x

Non definiamo qui formalmente tale relazione. Per noi è sufficiente sapere che due termini sono α-convertibili ogni volta che uno può essere ottenuto dall'altro semplicemente rinominando le variabili legate.

Ovviamente la convertibilità alfa mantiene completamente inalterato sia il significato del termine sia quanto questo significato sia esplicito. D'ora in poi, lavoreremo implicitamente su λ-termini modulo alfa conversione. Ciò significa che d'ora in poi due termini convertibili alfa come λz.z e λx.x sono per noi lo stesso termine.

Esempio:

(λz. ((λt. tz) z)) z

Tutte le variabili, tranne la z più a destra, sono vincolate, quindi possiamo rinominarle (applicare α-conversion). La z più a destra, invece, è libera. Non possiamo rinominarlo senza modificare il significato dell'intero termine.

È chiaro dalla definizione che l'insieme delle variabili libere di un termine è invariante per α-conversione, quello delle variabili legate ovviamente no.

La formalizzazione della nozione di calcolo in λ-calcolo: la teoria della β-riduzione

In un linguaggio funzionale calcolare significa valutare espressioni, rendendo sempre più esplicito il significato di una data espressione. Finora abbiamo formalizzato nel lambda-calculus le nozioni di programmi e dati (i termini nel nostro caso). Vediamo ora come formalizzare la nozione di calcolo. In particolare, la nozione di "passo computazionale di base".

Abbiamo notato nell'introduzione che nella valutazione di un termine, se c'è una funzione applicata ad un argomento, questa è sostituita dal corpo della funzione in cui il parametro formale è sostituito dall'argomento effettivo.

Quindi, se chiamiamo redex un qualsiasi termine della forma (λx.M)N, e chiamiamo M[N/x] il suo contratto, eseguire un passaggio computazionale di base su un termine P significa che P contiene un sottotermine che è un redex e che tale redex è sostituito dal suo contratto (la β-riduzione è applicata alla redex.)

La relazione tra redex e loro contratti è chiamata nozione di β-riduzione, ed è rappresentata come segue.

La nozione di β-riduzione

(λx.M) N →β M [N/x]

Un termine P β-si riduce di un passo a un termine Q se Q può essere ottenuto da P sostituendo un sottotermine di P della forma (λx.M)N con M[N/x]

Esempio:

Se avessimo la moltiplicazione nel nostro sistema, potremmo eseguire i seguenti calcoli

(λx. x\*x) 2 => 2\*2 => 4

qui il primo passo è una β-riduzione; il secondo calcolo è la valutazione della funzione di moltiplicazione. Vedremo come quest'ultimo calcolo possa effettivamente essere realizzato per mezzo di una sequenza di passi computazionali di base (e come anche il numero 2 e la moltiplicazione della funzione possano essere rappresentati da λ-termini).

A rigor di termini, M → N significa che M si riduce a N esattamente di un gradino di riduzione. Spesso ci interessa sapere se M può essere ridotto a N di un numero qualsiasi di passaggi.

scriviamo M —» N se M → M1 → M2 → ... → Mk ≡ N (K ≥ 0)

Significa che M si riduce a N in zero o più step di riduzione.

Esempio:

((λz.(zy))(λx.x)) —» y

Normalizzazione e possibilità di non risoluzione

Definizione

Se un termine non contiene β-redex, si dice che sia in forma normale.

Esempio:

λxy.y e xyz sono in forma normale.

Si dice che un termine ha una forma normale se può essere ridotto a un termine in forma normale.

Esempio:

(λx.x)y non è in forma normale, ma ha la forma normale y

Definizione:

Un termine M è normalizzabile (ha una forma normale, è debolmente normalizzabile) se esiste un termine Q in forma normale tale che

M —» D

Un termine M è fortemente normalizzabile, se non esiste un percorso di riduzione infinito fuori da M. Cioè, qualsiasi possibile sequenza di β-riduzioni alla fine porta a una forma normale.

Esempio:

(λx.y)((λz.z)t)

è un termine non in forma normale, normalizzabile e anche fortemente normalizzabile. Infatti le uniche due possibili sequenze di riduzione sono:

1. (λx.y)((λz.z) t) → y

2. (λx.y)((λz.z) t) → (λx.y) t) → y

Normalizzare un termine significa applicare riduzioni fino al raggiungimento di una forma normale (se presente). Molti λ-termini non possono essere ridotti alla forma normale. Un termine che non ha forma normale è

(λx.xx)(λx.xx)

Questo termine è solitamente chiamato Ω.

Sebbene diverse sequenze di riduzione non possano portare a diverse forme normali, possono produrre esiti completamente diversi: una potrebbe terminare mentre l'altra potrebbe "correre" per sempre. Tipicamente, se M ha una forma normale e ammette una sequenza di riduzione infinita, contiene un sottotermine L senza forma normale e L può essere cancellato da alcune riduzioni. La riduzione

(λy.a)Ω → a

raggiunge una forma normale, cancellando Ω. Ciò corrisponde a una trattazione degli argomenti chiamata per nome: l'argomento non viene ridotto, ma sostituito "così com'è" nel corpo dell'astrazione.

Il tentativo di normalizzare l'argomento genera una sequenza di riduzione non terminante:

(λy.a)Ω →(λy.a)Ω → ...

La valutazione dell'argomento prima di sostituirlo nel corpo corrisponde a un trattamento call-by-value dell'applicazione della funzione. In questi esempi, una strategia call-by-value non raggiunge mai la forma normale.

Alcuni importanti risultati della teoria della β-riduzione

La normale strategia di riduzione dell'ordine consiste nel ridurre, ad ogni passaggio, il β-redex più esterno a sinistra. Più a sinistra significa, ad esempio, ridurre L (se riducibile) prima di N in un termine LN. Più esterno significa, ad esempio, ridurre (λx.M) N prima di ridurre M o N (se riducibile). La normale riduzione degli ordini è una strategia di valutazione call-by-name.

Teorema (Standardizzazione)

Se un termine ha una forma normale, si raggiunge sempre applicando la normale strategia di riduzione degli ordini.

Teorema (confluenza)

Se abbiamo termini M, M1 e M2 tali che:

M —» M1 e M —» M2

allora esiste sempre un termine L tale che:

M1 —» L e M2 —» L

Cioè, in forma di diagramma:

Corollario (Unicità della forma normale)

Se un termine ha una forma normale, è unico.

Prova:

Supponiamo di avere un termine generico M, e supponiamo che abbia due forme normali N e N'. Significa che

M —» N e M —» N'

Applichiamo ora il teorema di confluenza. Allora esiste un termine Q tale che:

N —» Q e N' —» Q

Ma N e N' sono in forma normale, quindi non hanno alcuna redex. Quindi, significa necessariamente questo

N —» Q in 0 passi

e quindi che N ≡ Q.

Analogamente possiamo mostrare che N' ≡ Q. È quindi ovvio che N ≡ N'.

Il risultato di cui sopra è molto importante per i linguaggi di programmazione, poiché garantisce che se implementiamo lo stesso algoritmo in Scheme e in Haskell, le funzioni che otteniamo ci diano lo stesso risultato (se presente), anche se Scheme e Haskell sono funzionali linguaggi che utilizzano strategie di riduzione molto diverse.

Nella sintassi del λ-calcolo non abbiamo considerato tipi di dati primitivi (i numeri naturali, i booleani e così via), né operazioni primitive (addizione, moltiplicazione e così via), perché in effetti non rappresentano concetti realmente primitivi. Possiamo infatti rappresentarli utilizzando le nozioni essenziali già formalizzate nel λ-calcolo, ovvero variabile, applicazione e astrazione.

La teoria della β-conversione

È possibile formalizzare la nozione di equivalenza computazionale dei programmi nel calcolo lambda. Ciò viene fatto introducendo la relazione di β-conversione tra λ-termini, indicata con M =β N

Informalmente, potremmo dire che due termini sono β-convertibili quando "rappresentano la stessa informazione": infatti se uno può essere trasformato nell'altro eseguendo zero o più β-riduzioni e β-espansioni.

(Un'espansione β è l'inverso di una riduzione: P[Q/x] ← (λx.P)Q ) .

Nel nostro corso non tratteremo la teoria della β-conversione.

Rappresentazione di tipi di dati e funzioni su di essi

Nella sintassi del λ-calcolo non abbiamo considerato tipi di dati primitivi (i numeri naturali, i booleani e così via), né operazioni primitive (addizione, moltiplicazione e così via), perché in effetti non rappresentano concetti realmente primitivi. Possiamo infatti rappresentarli utilizzando le nozioni essenziali già formalizzate nel λ-calcolo, ovvero variabile, applicazione e astrazione (non forniamo qui alcun esempio, ma ci riferiamo a [PS]). Inoltre, è possibile rappresentare nel lambda-calculus qualsiasi funzione calcolabile (notare che non usiamo termini lambda solo per definire funzioni matematiche, ma per rappresentare algoritmi particolari per calcolare funzioni matematiche).

Rappresentazione di algoritmi ricorsivi

Nella sintassi del λ-calcolo non c'è alcuna operazione primitiva che ci permetta di dare nomi ai termini lambda. Tuttavia, quella di dare nomi alle espressioni sembra essere una caratteristica essenziale dei linguaggi di programmazione per definire i programmi mediante ricorsione, come in Scheme

(definisci fatto (lambda (x) (if (= x 0) 1 (\* x (fatto (- x 1))))))

o in Haskell

fatto 0 = 1

fatto n = n \* fatto (n-1)

Quindi, per definire algoritmi ricorsivi nel λ-calcolo, sembrerebbe essenziale poter dare nomi alle espressioni, in modo da potervi riferire ricorsivamente ad esse.

Abbastanza sorprendentemente, tuttavia, non è affatto vero! Infatti, possiamo rappresentare funzioni ricorsive nel lambda-calcolo utilizzando i pochi ingredienti che abbiamo già a portata di mano, sfruttando il concetto di punto fisso.

La nozione di punto fisso può essere rappresentata nel calcolo lambda attraverso termini come

λf. (λx.f(xx)) (λx.f(xx))

(questo termine è tradizionalmente chiamato Y)

Infatti, per ogni termine M si ha che M(YM)=βYM

Se consideriamo M una funzione (poiché possiamo applicarle dei termini), il termine YM è allora un punto fisso di M (un punto fisso di una funzione f:D->D essendo un elemento d di D tale che f (d)=d).

Da punti fissi possiamo derivare la nozione di ricorsione nel calcolo lambda: infatti, intuitivamente, qual è la funzione fattoriale? è la funzione tale che

fattoriale = λx. se (x=0) allora 1 altro (x\*fattoriale(x-1))

ciò significa che la funzione fattoriale è il punto fisso di

λf. λx. se (x=0) allora 1 altro (x\*f(x-1))

Quindi, algoritmi ricorsivi possono essere rappresentati nel calcolo lambda per mezzo di punti fissi (e quindi usando termini come Y, chiamati combinatori a virgola fissa.) In [PS] viene discusso più precisamente come usare combinatori a virgola fissa in per rappresentare algoritmi ricorsivi. In particolare utilizzando un combinatore diverso da Y, ovvero il combinatore Θ.

Dopo aver sistemato i combinatori a virgola fissa in [PS] si potrebbe verificare che anche il seguente termine (combinatore di Klop) sia in realtà un combinatore a virgola fissa:

£££££££££££££££££££££££££££

dove £ è il termine lambda

λabcdefghijklmnopqstuvwxyzr.(r(questoèuncombinatoreavirgolafissa))

Strategie di riduzione

Uno degli aspetti che caratterizza un linguaggio di programmazione funzionale è la strategia di riduzione che utilizza. Informalmente una strategia di riduzione è una "politica" che sceglie un β-redex, se presente, tra tutti i possibili presenti nel termine. È possibile formalizzare una strategia in funzione, anche se non lo facciamo qui.

Vediamo, ora, stesse classi di strategie.

Strategie call-by-value

Queste sono tutte le strategie che non riducono mai un redex quando l'argomento non è un valore.

Ovviamente occorre formalizzare con precisione ciò che si intende per "valore".

Nell'esempio seguente consideriamo un valore un termine lambda in forma normale.

Esempio:

Supponiamo di avere:

((λx.x)((λy.y) z)) ((λx.x)((λy.y) z))

Una strategia call-by-value non riduce mai il termine (λx.x)((λy.y) z) perché l'argomento non è un valore. In una strategia call-by-value, quindi, possiamo ridurre solo uno dei redex sottolineati.

Scheme utilizza una specifica strategia call-by-value e una particolare nozione di valore.

Informalmente, finora abbiamo considerato come "valori" i termini in forma normale. Per Schema, invece, anche una funzione è considerata un valore (un'espressione come λx.M, in Schema, è un valore).

Allora, se lo abbiamo

(λx.x) (λz.(λx.x) x)

La strategia di riduzione di Scheme prima riduce il redex più grande (l'intero termine), poiché l'argomento è considerato un valore (una funzione è un valore per Scheme). Dopo la riduzione l'interprete si ferma, poiché il termine λz.((λx.x) x) è un valore per Schema e quindi non può essere ulteriormente valutato.

Strategie call-by-name

Queste sono tutte le strategie che riducono un redex senza verificare se il suo argomento è un valore o meno. Una delle sue sottoclassi è l'insieme delle strategie pigre. Generalmente nei linguaggi di programmazione una strategia è pigra se è call-by-name e riduce un redex solo se è strettamente necessario per ottenere il valore finale (anche questa strategia è chiamata call-by-need). Una possibile strategia chiamata per nome è quella più esterna a sinistra: prende, nell'insieme del possibile redex, quella più a sinistra e quella più esterna.

Esempio:

Indichiamo con I la funzione identità e consideriamo il termine:

(I (II)) ( I (II))

la strategia più esterna a sinistra prima riduce la redex sottolineata.

Abbiamo visto prima che la strategia più esterna a sinistra è molto importante poiché, per il teorema di standardizzazione, se un termine ha una forma normale, lo otteniamo seguendo tale strategia. Il problema è che il numero di passaggi di riduzione a volte è maggiore di quelli strettamente necessari.

Se una lingua utilizza una strategia call-by-value, dobbiamo stare attenti perché se un'espressione non ha una forma normale, la ricerca potrebbe continuare all'infinito.