Lambda Calculus

Tutto quello che avreste voluto sapere sul lambda-calcolo …

… ma non avete mai osato chiedere

Hilbert:
fondamenti della matematica
e… l’informatica

David Hilbert

Entscheidungsproblem

Il problema della decisione

E’ possibile definire una procedura, eseguibile del tutto meccanicamente, in grado di stabilire, per ogni formula espressa nel linguaggio formale della logica del primo ordine, se tale formula è o meno un teorema della logica del primo ordine.

Come si formalizza la nozione di procedura effettiva (algoritmo)? Diverse risposte

Alonzo Church: Lambda calculus

An unsolvable problem of elementary number theory, *Bulletin the American Mathematical Society*, May 1935

Come si formalizza la nozione di procedura effettiva (algoritmo)? Diverse risposte

Kurt Gödel: Recursive functions

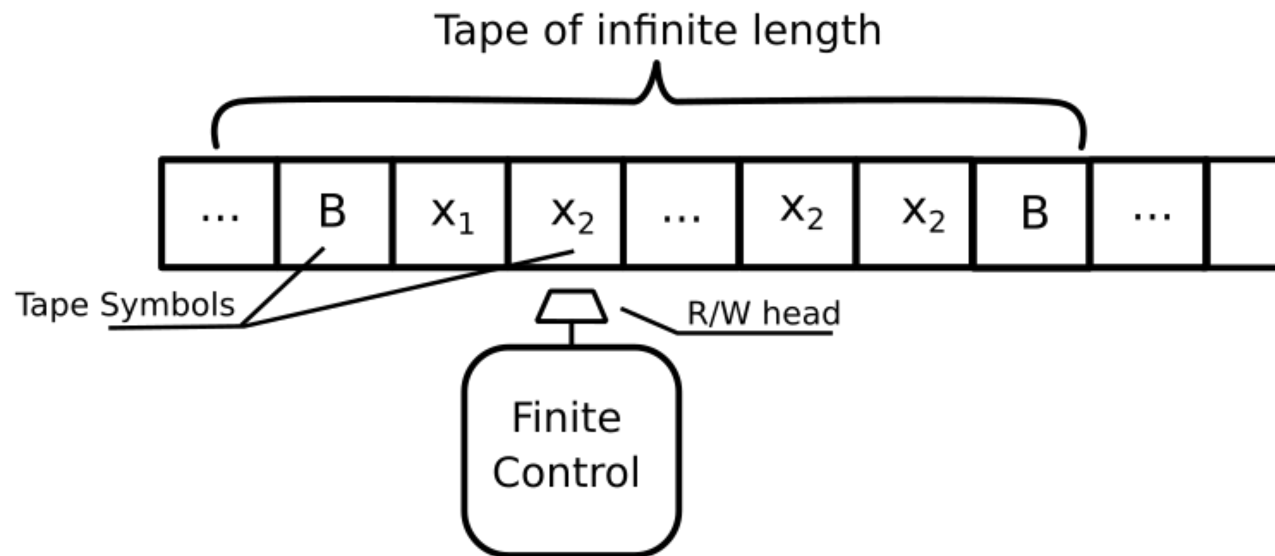
Stephen Kleene, General recursive functions of natural numbers, *Bulletin the American Mathematical Society*, July 1935

Come si formalizza la nozione di procedura effettiva (algoritmo)? Diverse risposte

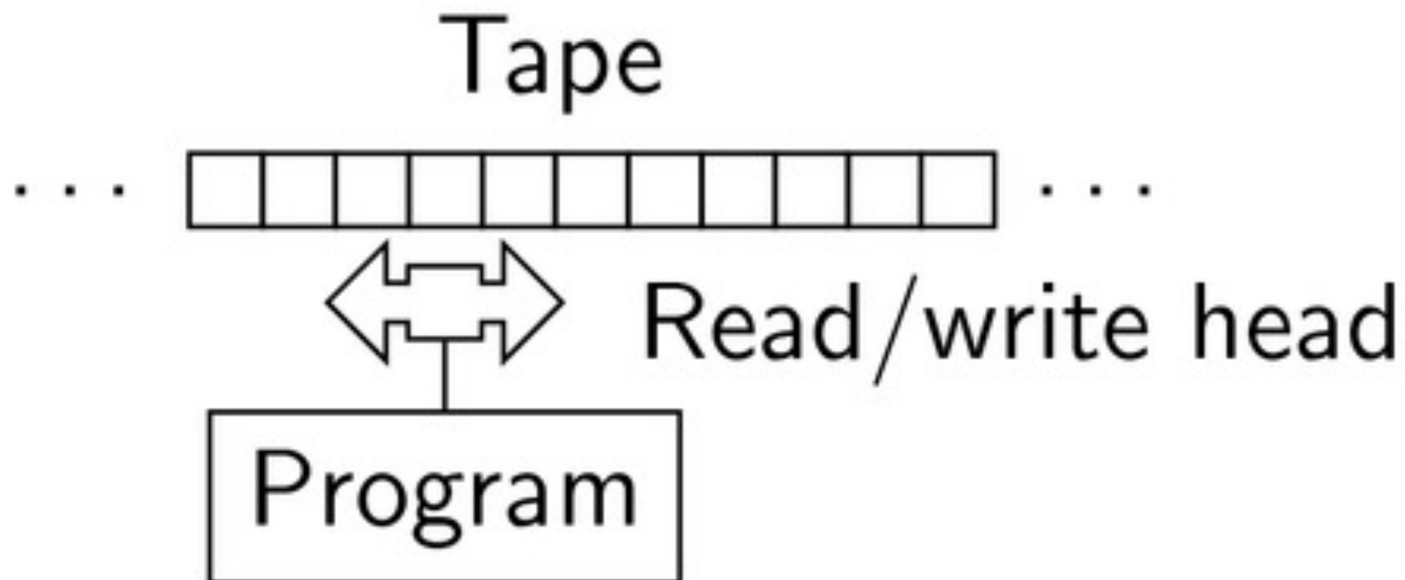
Alan M. Turing: Turing machines

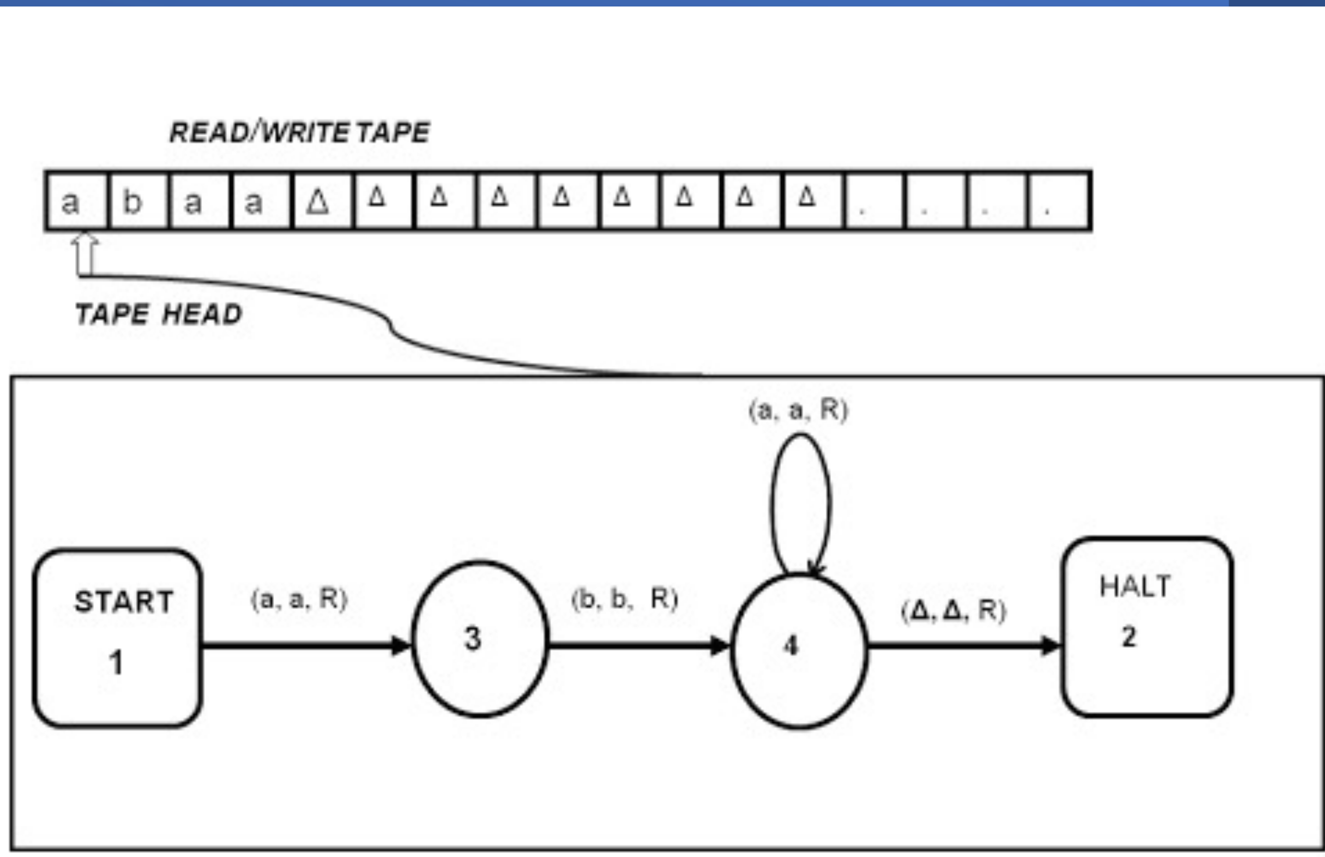
On computable numbers, with an application to the *Entscheidungsproblem*, *Proceedings of the London Mathematical Society*, received 25 May 1936

Dalla Macchina di Turing (ripasso)



ai programmi memorizzati





A Turing Machine for aba*

Macchina di Turing Universale (UTM)

La macchina di Turing universale (UTM) è una macchina di Turing capace di **simulare il comportamento di una qualunque macchina di Turing**.

La UTM è stata definata da Turing nel suo fondamentale lavoro del 1936 e gli ha consentito di dare una risposta negativa al problema della decidibilità, "Entscheidungsproblem”.

Turing completeness

Un linguaggio di programmazione è detto **Turing completo** se è in grado di *calcolare qualsiasi funzione calcolabile da una macchina di Turing*

Come si fa a dimostrare che un linguaggio di programmazione L è Turing completo?

- *Possiamo associare a ogni macchina di Turing un programma scritto in L: il programma simula il comportamenti della macchina di Turing*
- *possiamo trasformare (compilare) un qualsiasi programma scritto in un linguaggio Turing-completo in un programma equivalente scritto in L*

Turing completeness (come implementazione di linguaggi di programmazione)

- Un linguaggio di programmazione è detto **Turing completo** se è in grado di *calcolare qualsiasi funzione calcolabile da una macchina di Turing*
- *Come si fa a dimostrare che un linguaggio di programmazione L è Turing completo?*

1. *Possiamo scrivere in L un **interprete** della Macchina di Turing Universale*
2. *Possiamo scrivere un **compilatore** che tradude un qualsiasi programma scritto in un linguaggio Turing-complete in un programma equivalente scritto in L*

Dalle Macchine di Turing alla macchina di von Neumann

- Il modello di caclo delle machine di Turing ha ispirato **Von Neumann** che ha definito la base della struttura dei computer attuali
- Architettura von Neumann: due componenti principali
  - **Memoria**, dove sono memorizzati i programmi e i dati
  - **Unità centrale di elaborazione**, che ha il compito di eseguire i programmi immagazzinati in memoria prelevando le istruzioni (e i dati relativi), interpretandole ed eseguendole una dopo l’altra

(Turing)-van Neumann programming languages

- Un linguaggio di programmazione nello stile (Turing-)Von Neumann è un linguaggio di programmazione che prevede astrazioni di programmazione che riproducno ad alto livello la struttura della architetture di von Neumann
- **variabili <astrazione> celle di memoria (MdT tape)**
- **istruzioni di controllo <astrazione> istruzioni di test&jump (MdT control)**
- **assegnamento <astrazione> modifica dello stato (fetching & storing instructions)**

Turing-van Neumann Languages

1957 – FORTRAN
1959 – ALGOL
1962 – SIMULA

1972 – C

1979 – C++

1991 – Python
1995 – Java

Can programming be liberatwd from the von neumann style?

Comm. ACM 21 (8) 1978

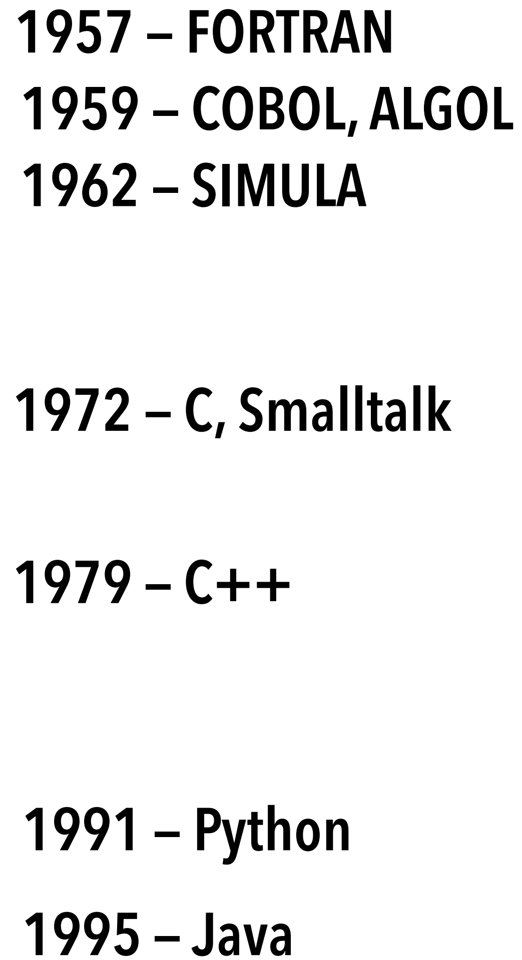
Backus

- John Backus ha osservato che la nozione di assegnamento nei linguaggi di von Neumann divide la programmazione in due mondi.
- Il primo mondo consiste di **espressioni**, uno spazio matematico ordinato con proprietà algebriche potenzialmente utili: la maggior parte dei calcoli avviene nel mondo delle espressioni
- Il secondo mondo è lo spazio delle **istruzioni**, uno spazio matematico disordinato con poche proprietà utili (la programmazione strutturata può essere vista come un'euristica limitata che cerca di porre un minimo di ordine a questo spazio)

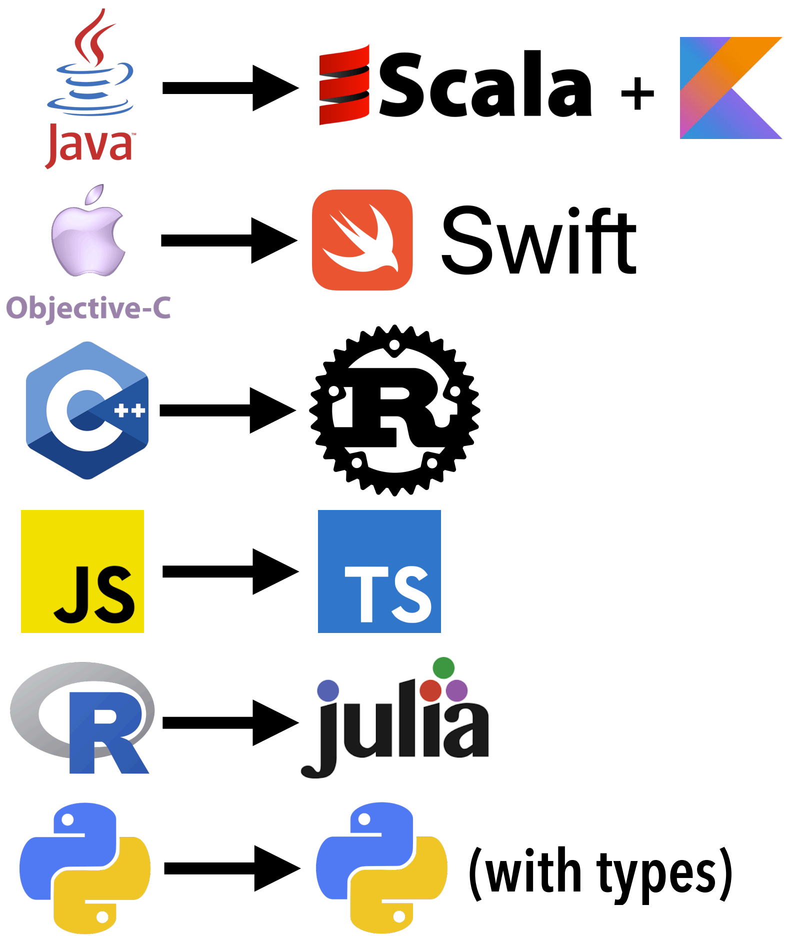
Programmazione funzionale

- La **programmazione funzionale** è un paradigma di programmazione in cui il flusso di esecuzione del programma assume la forma di una serie di valutazioni di funzioni matematiche.
- Il punto di forza principale di questo paradigma è la mancanza di effetti collaterali (*side-effect)* delle funzioni, il che comporta una più facile verifica della correttezza (assenza di bug) del programma e la possibilità di una maggiore ottimizzazione dello stesso.
- Il Lambda calcolo (Church 1935) può essere considerato il primo linguaggio di programmazione funzionale.

Turing languages



Church languages



Una prospettiva di evoluzione

Alcune considerazioni personali (1)

Il mondo del software è pronto per linguaggi di programmazione migliori

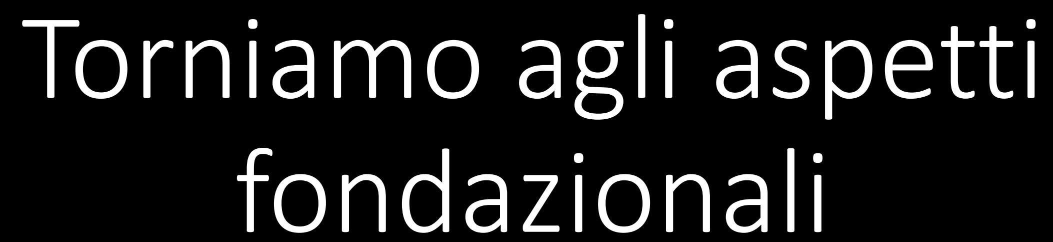
I linguaggi che usiamo oggi sono cambiati a malapena in 60 anni

Gli sviluppatori passano gran parte del tempo a scrivere ecosistemi software enormi, pieni di bug e poco sicuri

Considerazioni personali (2)

I linguaggi di programmazione funzionali si fondano su modelli formali di programmazione

I sistemi di tipi promuovono la modularità e la verifica di proprietà di programmni


Torniamo agli aspetti fondazionali

Tesi di Church-Turing

La tesi di Church-Turing è un'ipotesi che afferma: “**Se un problema è calcolabile, allora esisterà una macchina di Turing in grado di risolverlo (cioè di calcolarlo)**”.

Formalmente: la classe delle funzioni calcolabili coincide con quella delle funzioni calcolabili da una macchina di Turing.

Le funzioni Ricorsive Generali (Kleene-Goedel), il Lambda Calcolo (Chrurch) sono modelli di calcolo Turing equivalenti: computano le stesse funzioni di una macchina di Turing.

Tesi di Church-Turing

- Modelli di calcolo meno potenti di una macchina di Turing:
  - **espressioni regolari (automi a stati finiti)**
  - **macchine che terminano sempre (funzioni totali)**
- Considerazione: non è noto modelo di calcolo più potente della macchina di Turing in termini computazionali.
  - Quindi tutto ciò che non è calcolabile dalla TM non può essere calcolato da nessun altro formalismo **a noi noto**.

Lungo preambolo …
Quale è il motivo?

Linguaggi di programmazione: alcune domande

- Quali caratteristiche di un linguaggio linguaggio sono necessarie e indispensabili per esprimere tutte le funzioni calcolabili?
- E’ possible definire il nucleo di un linguaggio di programmazione che è Turing completo?

Analisi derivata dall’esperienza nella programmazione

Funzioni con diversi parametri formali

```
function sum(a, b) {
  return a * b; }
```

Utlizziamo una trasformazione, currying, che traduce una funzione con diversi parametri in una sequenza di funzioni che prendono ciascuna un singolo argomento.

```
res = sum(a, b)

h = g(a)
res = h(b)

res = sum (a) (b)
```

*la funzione **sum** da **Int x Int -> Int** viene traformata in una funzione **g** che prende in ingresso un intero e restituisce come risultato una funzione da interi a interi **g: Int -> (Int -> Int)***

Analisi derivata dall'esperienza nella programmazione

Funzioni con diversi parametri formali

```
function sum(a, b) {
  return a * b; }
```

LE FUNZIONI CON DIVERSI PARAMETRI FORMALI
SONO UTILE
ZUCCHERO SINTATTICO!!!

```
h = g(a)
res = h(b)

res = sum (a) (b)
```

*la funzione **sum** da **Int x Int -> Int** viene traformata in una funzione **g** che prende in ingresso un intero e restituisce come risultato una funzione da interi a interi **g: Int -> (Int -> Int)***

Intermezzo: Haskell Curry

- L’operatore di trasformazione è detto ***currying***, in onore di Haskell Curry, che ne studiò a lungo le proprietà.
- Nei linguaggi di programmazione in currying risulta di fatto l'applicazione parziale di una funzione ai suoi argomenti: ogni funzione di più argomenti si considera un funzionale e gli argomenti vengono presi appunto "uno alla volta" a cominciare dal primo dando come risultato altri funzionali o (arrivati al penultimo argomento) una funzione.

Currying in Javascript

```
function curry(f) {
    return function(a) {
        return function(b) {
         return f(a, b);
        };
     };
}

//uso
function sum(a, b) { return a + b; }

let curriedSum = curry(sum);
alert( curriedSum(1)(2) );
// 3
```

Il lambda calcolo

Sintassi

Un programma è una **espressione**

$e ::= x$	**variabile**
$\mid \lambda x.e$	**astrazione funzionale (fun dec)**
$\mid e\ e$	**applicazione (fun call)**

Niente altro!!!
La sintassi è terminata

Nota: da dove viene fuori il λ?

Nella prime note Church utilizzava “hat”

$$\hat{x}.\ \text{x+1}$$

ma una notazione lineare era più semplice

$$\wedge\, x. x + 1$$

.. che portava direttamente a usare la lettera greca Λ

$$\Lambda x. x + 1$$

… ma per evitare confusione con la A maiscola

$$\lambda x. x + 1$$

La sintassi in modo formale

- Dato un insieme di variabili Var = x, y, z, …… La sintassi delle espressioni (lambda) Exp risulta:

$$\frac{x \in Var}{x \in Exp}$$

$$\frac{x \in Var \quad e \in Exp}{\lambda x. e \in Exp}$$

$$\frac{e_1 \in Exp \quad e_2 \in Exp}{e_1 e_2 \in Exp}$$

La sintassi in modo alternativo

Var x : x è una variabile

Exp e: e è una espressione lambda

$$\frac{Var\ x}{Exp\ x}$$

$$\frac{Var\ x \quad Exp\ e}{Exp\ \lambda x.e}$$

$$\frac{Exp\ e_1 \quad Exp\ e_2}{Exp\ e_1\ e_2}$$

Lambda expressioni: funzioni anonime

- L'espressione $\lambda x.e$ è una funzione anonima (funzione a cui non viene associato alcun none)
  - Il nome x è la dichiarazione del parametro della funzione anonima

- Funzioni anonime (Javascript)
  - **function(a){ return a + 1; }**
- Funzioni anonime (OCaml)
  - **fun x -> x+1**
- Funzioni anonime aka Lambda (Java)
  - **(int x) -> x + 1**

Lambda espressioni: applicazione

- La lambda espressione **e1 e2** l'espressione della funzione **e1** è applicata all'espressione dell'argomento **e2**.
- L'applicazione corrisponde alla chiamata di funzione in Javascript (invovazione di un metodo in Java):
  - l'espressione **e1** definisce la funzione
  - l'espressione **e2** definisce il parametro attuale.
- Il calcolo λ è molto generale e permette alle definizioni di funzione di apparire direttamente nelle chiamate di funzione.

Convenzioni sintattiche

- Costruttore lambda: lo scope del lambda si estende il più a destra possible
  - **λx. λy. x y** è la stessa cosa di **λx. (λy. (x y))**
- L'applicazione funzionale associa a sinistra
  - **e1 e2 e3 è zucchero sintattico di (e1 e2) e3**
- Per semplificare e rendere maggiormente intuitivi gli esempi utilizzeremo nella scrittura di lambda espressioni con costanti prese da un un insieme **Const** (Const = {0, 1, 2, ... }. Inoltre utilizzeremo operazioni aritmentiche standard come +, *, ...

Intermezzo: alberi di sintassi astratta

- A volte è utile visualizzare una lambda espressione come a un albero tecnicamente ***alberi di sintassi astratta***, ***abstract syntax tree***), la cui struttura corrisponde al modo in cui l'espressione si costruisce seguendo le regole della grammatica.
- Alberi di sintassi astratta: albero che rappresenta la struttura sintattica di una programma in accordo regole grammaticali del linguaggio.

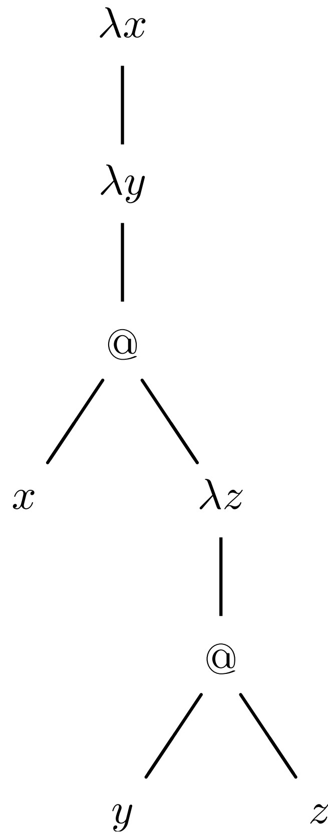
L'albero di sintassi astratta di una **variabile x** è l’albero che consiste della foglia, etichettata **x**.

L'albero di sintassi astratta di **un'astrazione λx.e** è l’albero che consiste in un nodo etichettato **λx** con un solo sottoalbero, che è l'albero di sintassi astratta di **e**.

L'albero di sintassi astratta di **un'applicazione e e’** consiste in un nodo etichettato **@** con due sottoalberi:

il sottoalbero sinistro è l'albero di sintassi astrata di **e,**

il sottoalbero destro è l'albero di sintassi astratta di **e’**.



L'albero di sintassi astratta deell'espressione
λx. λy. (x (λz. y z))

λ: operatore di binding

- Il λ ($\lambda x.e$) è un operatore che lega le variabili, nello stesso modo in cui la variabile x nella formula $\forall x.P$ della logica del primo ordine (dove la sottoformula P può contenere x) è una variabile fittizia che potrebbe essere rimpiazzata con qualsiasi altra variabile *fresca*,
- Una variabile fresca è un nome nuovo: un nome che non compare da nessuna parte in nessuno delle espressioni che stiamo trattando.

Variabile libere e legate

Le variabili in una lambda espressione che sono introdotte (dichiarate) in un qualche λ sono **legate** da quel λ.

Tutte le altre variabili che non sono associate a una dichiarazione con un qualche λ sono **libere**.

ESEMPI

λn. n+1	**n è legata**
λx. λy. (x*y*z)	**x e y sono legate z è libera**
(λf.f(p+q))(sqrt)	**f è legata, p, q e sqrt sono libere**

Variabili libere: definizione formale

L'insieme delle variabili lirbere, **FV**, di una lambda espressione è definito dall'equazioni

$$\mathbf{FV(x) = \{x\}}$$
$$\mathbf{FV(e1\ e2) = FV(e1) \cup FV(e2)}$$
$$\mathbf{FV(\lambda x.e) = FV(e) \setminus \{x\}}$$

Alpha equivalenza

L espressioni λa.(a + 1) e λb.(b + 1) sono detti α-equivalenti.

La sostituzione di una variabile legata con
un'altra *opportuna* variabile fresca costituisce una traformazione sintattica
di programmi che solitamente (in matematica) viene chiamata α-conversione

function(a){ return a + 1; }
è α-equivalente a
function(b){ return b + 1;

Espressioni α-equivalenti rappresentano lo stesso programma.
.

La nozione di esecuzione

Cosa significa ***eseguire*** (***valutare***) una lambda espressione?

La valutazione, **eval**, di lambda espressioni comporta solamente la chiamata di funzioni.

La valutazione di ((λx.e1) e2))

eval((λx.e1) e2))

consiste nel valutare e1 dove ogni occorrenza di x in e1 è rimpiazzata da e2

Intuizione (informatica)
La chiamata della funzione anonima λx.e1 con parametro attuale e2 corrisponde ad eseguire il corpo e1 della funzione dopo una trasformazione sintattica: ogni occorrenza del parametro
formale x è sostituita con l'espressione parametro attuale e2

La valutazione di ((λx.e1) e2))

eval((λx.e1) e2))

consiste nel valutare e1 dove ogni occorrenza di x in e1 è rimpiazzata da e2

Intuizione (informatica)

La chiamata della funzione anonima λx.e1 con parametro attuale e2 non prevede la valutazione del parametro attuale (come avviene nella call-by-value).

Il passaggio del parametro comporta una traformazione sintattica:ogni occorrenza del parametro formale x è sostituita con l'espressione parametro attuale e2

La valutazione di ((λx.e1) e2))

eval((λx.e1) e2))

consiste nel valutare e1 dove ogni occorrenza di x in e1 è rimpiazzata da e2

Intuizione (informatica)

La chiamata della funzione anonima λx.e1 con parametro attuale e2 non prevede la valutazione del parametro attuale.

E' una forma di valutazione lazy che consiste nel ritardare la computazione relativa alla valutazione del parametri finché il valore (del parametro) non è richiesto.

Intermezzo: passaggio parametri (1)

- **Quale è la regole per la valutazione dell'applicazione di funzione?**
- Il primo passo consiste nella valutazione dell'espressione della funzione:
- Il secondo passo definisce il meccanismo di passaggio dei parametri. Abbiamo due alternative:

  1. tutte le occorrenze della variabile legata alla funzione nell'espressione del corpo della funzione sono sostituite dal valore dell'espressione parametro formale.

  2. tutte le occorrenze della variabile legata alla funzione nell'espressione del corpo della funzione sono sostituite dall'espressione non valutata che definisce il parametro attuale

- Dopo aver effettuato il passaggio dei parametri si passa a valutare il corpo della funzione.

Passaggio parametri (2)

Il primo approccio è chiamato *ordine applicativo*, applicative order, ed è sostanzialmnete la regola 'call by value': l'espressione del parametro attuale è valutata prima di essere associato al parametro formale.

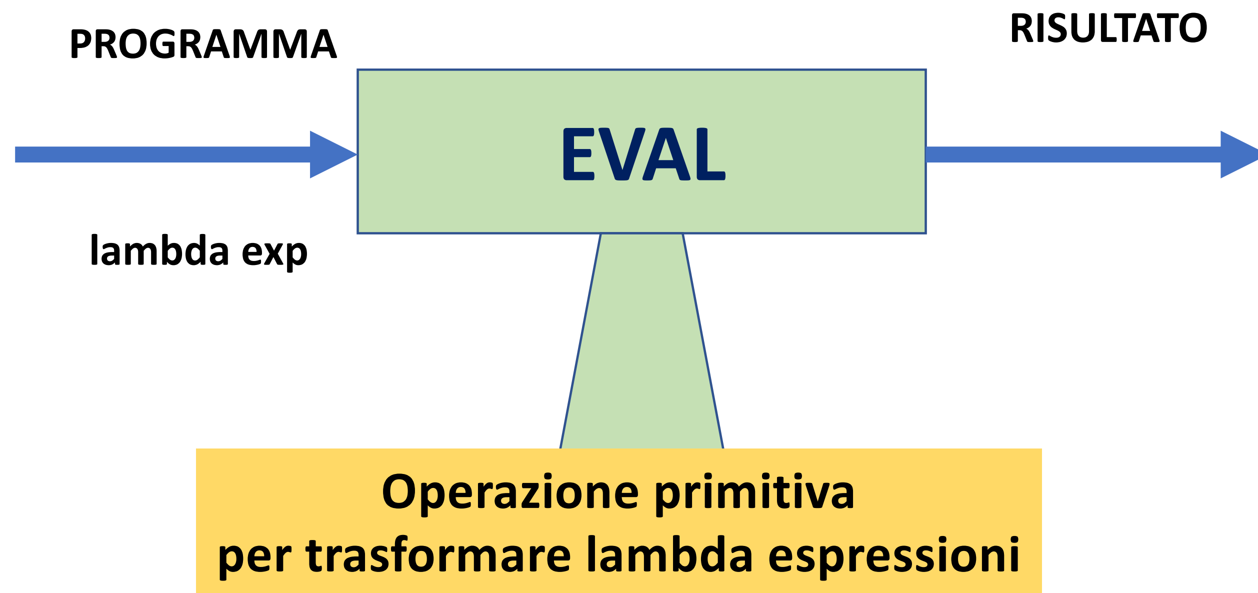
Il secondo approccio è chiamato *ordine normale,* normal order, ed è un meccanismo lazy di passaggio dei parametri, come la call-by name' in ALGOL 60: l'espressione del parametro attuale non viene valutata prima di essere associata al parametro formale.

Eseguire lambda espressioni

Come è strutturata una macchina in grado di valutare lambda espressioni?

Eseguire lambda espressioni

Come è strutturata una macchina in grado di valutare lambda espressioni?



Sostituzione

La notazione **e1{x:=e2}** è introdotta per descrivere la lambda espressione **e1** in cui ogni occorrenza della variabile **x** è sostituita dalla lambda espressione **e2**.

Problema

Se **e2** contiene una variabile libera **y** che è legata in **e1** c'è il rischio che **y** venga catturata dal legatore λ di **e1**.

Legare una variabile libera non è una bella idea!!!.

Soluzione: si deve α-convertire prima e1 introducendo una variabile fresca per ottenere una sostituzione che eviti il problema di legare una variabile libera (capture-avoiding substitution).

Notazione alternativa

La notazione

$$e1\{e2/x\}$$

è spesso utilizzata per definire l'operazione di sostituzione

Capture-avoiding substitution: esempio

$(\lambda x.(x*y))\ \{y:=(x+x)\}$ --- α-conversion, z fresh

$(\lambda z.(z*y))\ \{y:=(x+x)\} =$

$(\lambda z.(z*(x+x)))$

Capture-avoiding substitution: definizione

x{x:=e} ≡ e,

y{x:=e} ≡ y se x != y,

(e1 e2) {e/x} ≡ (e1 {x:=e})(e2 {x:=e}),

(λy.e1){x:=e} ≡ λy.(e1 {x:=e}) se
y != x e y ! ∈ FV(e)

(λy.e1){x:=e} ≡ λz.((e1 {y:=z}){x:=e}) se
y != x e y ∈ FV(e) e z fresca

Beta reduction

La regola di riduzione β (β -reduction) è la regola fondamentale che definisce il lambda calcolo come modello di calcolo universale

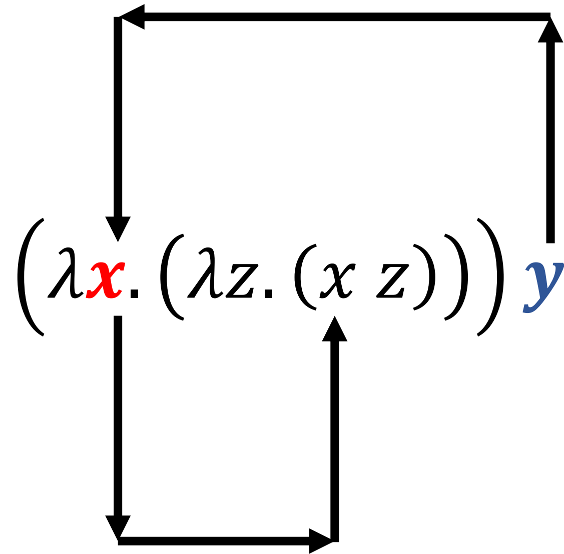
$$(\lambda\, x. e_1)e_2 \rightarrow e_1\{x \coloneqq e_2\}$$

La regola di β –reduction cattura precisamente la nozione di applicazione funzionale

Esempio

Parametro formale

Parametro attuale



$\lambda z (y\ z)$

Beta Reduction

$$(\lambda x. e_1) e_2 \to e_1 \{x := e_2\}$$

$$e_1 = \lambda z. (x\ z) \qquad e_2 = y$$

La regola di beta reduction corrisponde ad un passo di calcolo, che restituisce il corpo della funzione e1 dove il parametro attuale e2 viene sostituito al parametro formale x della funzione.

La sostituzione implementa pertanto il meccanismo di passaggio del parametro nella modalità per nome (lazy).

Interprete del lambda calcolo

La regola di beta reduction permette di definire l'interprete del lamba calcolo evidenzianto il meccanismo di **passaggio dei paramentri**

eval((λx.e1) e2)) = eval(**e1**{**x:=e2**})

Esempi: funzione identità

$(\lambda x. x)\ z \rightarrow z$

$(\lambda x. x)\ (\lambda y. y) \rightarrow (\lambda y. y)$

Il lambda calcolo permette di scrivere in modo naturale funzioni che possono prendere altre funzioni come parametri e/o restituire funzioni come risultato

Esempio

$$(\lambda x.\, x\ y)z \rightarrow zy$$

\\ Una funzione che prende come argomento una funzione e la applica a y

$$(\lambda x.\, x\ y)(\lambda z.\, z) \rightarrow (\lambda \mathrm{z}.\, z)y \rightarrow y$$

\\ paramero attuale è la funzione identità

Data una lambda espressione possiamo avere più di una scelta per applicare le regola di β-riduzione: le riduzioni possono essere applicate ovunque in una lambda espressione

$$(\lambda x.\, x + x)\big((\lambda y.\, y)5\big) \rightarrow \big((\lambda y.\, y)5\big) + ((\lambda y.\, y)5)$$

$$(\lambda x.\, x + x)\big((\lambda y.\, y)5\big) \rightarrow (\lambda x.\, x + x)5$$

La nozione di riduzione … in modo formale

La **β riduzione**, →, è la relazione definita dalle regole seguenti

$$(\lambda x.e_1)e_2 \to e_1\{x := e_2\}$$

$$\frac{e_1 \to e'}{e_1 e_2 \to e' e_2}$$

$$\frac{e_2 \to e'}{e_1 e_2 \to e_1 e'}$$

$$\frac{e \to e'}{\lambda x.e \to \lambda x.e'}$$

Beta riduzione

Con $\Longrightarrow$ indichiamo la chiusura riflessiva e transitiva di $\rightarrow$.

Questo equivale all'aggiunta di entrambe le regole di riflessività e transitività.

La lambda espressione e_1 è β-riducibile alla lambda espressione $\mathbf{e_2}$ quando vale che $\mathbf{e_1 \Longrightarrow e_2}$

Valori calcolati

- Quando una lambda espressione non può essere ridotta ulteriormente diciamo che è in **forma normale beta**
  - **Una lambda espressione è in forma normale quando non è più riscrivibile per mezzo della regola di beta riduzione. Dato che la beta riduzione rappresenta un passo di calcolo, allora la sua chiusura transitiva ne rappresenta una qualsiasi computazione**.
- Una lambda espressione ridotta in forma normale rappresenta pertanto il risultato finale, **il valore calcolato**, della computazione.

- **Valori: esempi**
- $\lambda x. x$ **è un valore (le funzioni sono valori)**
- $\lambda t. \lambda f. t$ **è un valore**

Beta equivalenza

- La relazione di riduzione $\Longrightarrow$ fornisce le basi per poter definire una nozione di equivalenza tra lambda espressioni.
- Due lambda espressioni e_1 , e_2 si dicono **beta equivalenti**, $e_1 \equiv_\beta e_2$ se
  - *Sono identiche (modulo alpha conversione)*
  - $e_1 \Longrightarrow e_2$ *oppure* $e_2 \Longrightarrow e_1$
  - $e_1 \Longrightarrow e$ *e* $e_2 \Longrightarrow e$
- Esempio
- $(\lambda x. x)z$ è beta equivalente a $(\lambda x.\ \lambda y. x)z\ w$ dato che entrambe sono risotte alla stessa lamnda espressione z.

Beta equivalenza

Intuizione:

due lambda espressioni sono beta equivalenti quando sono indistinguibili dal punto di vista computazionale: calcolano gli stessi risultati.

Teorema di Church Rosser

Il Teorema di Church – Rosser afferma che l'ordine in cui vengono scelte le beta riduzioni non influisce sul risultato finale. Più precisamente, se ci sono due riduzioni distinte o sequenze di riduzioni che possono essere applicate alla stessa lambda espressione, allora esiste una lambda espressione che è raggiungibile da entrambi i risultati, applicando sequenze (possibilmente vuote) di riduzioni aggiuntive.

L’ordine in cui vengono applicate le riduzioni non influisce sul risultato finale.

Non terminazione

$$\Omega = (\lambda x.\, x\ x)(\lambda x.\, x\ x)$$

La lambda espressione Ω non può essere ridotta in forma normale

Contiene una sola espressione riducibile tramite beta riduzione.
La riduzione produce ancora Ω

$$\Omega = (\lambda x.\, x\ x)(\lambda x.\, x\ x) \rightarrow (\lambda x.\, x\ x)(\lambda x.\, x\ x) \rightarrow \ \ldots.$$

Funzioni di ordine superiore

Il lambda calcolo è più potente di quanto la sua definizione minimale possa suggerire.

Non è possible definire funzioni con più argomenti, tuttavia è facilissimo descrivere il comportamento di funzioni con argomenti multipli utilizzazzando funzioni di ordine superior (higher order functions) ovvero funzioni che producono come risultato del calcolo altre funzioni

Esempio

$$F = \lambda(x, y).e$$

$$F(e_1, e_2) \rightarrow e\{x := e_1\}\{y := e_2\}$$

$$F = \lambda x.\lambda y.e$$

$$(F\ e_1)e_2 \rightarrow (\lambda y.e)\{x \coloneqq e_1\}e_2 \rightarrow e\{x := e_1\}\{y := e_2\}$$

Funzione che invocata con il parametro atttuale e1 produce come risultato una funzione che invocata con parametro attuale e2 produce il risultato richiesto

Currying

La trasformazione di funzioni a più argomenti con funzioni di ordine superiore si chiama currying dopo l'uso fatto dal logico Curry.

La trasformazione è stata introdotta originariamente dal logico Shönfinkel …. Ma shönfinkeling non suona benissimo!!!

… e la ricorsione?

Nel lambda calculo, non è previsto un meccanismo primitive un per definire funzioni ricorsive.

Possiamo definire funzioni ricorsive nel lambda calcolo?

Il combinatore Y è una funzione di ordine superiore usata per implementare la ricorsione in qualsiasi linguaggio di programmazione che non la supporti nativamente.

È stato introdotto dal matematico e logico Haskell Curry negli anni '40 ed è considerato una delle idee più belle nella programmazione e nella logica.

Il combinatore Y

$$Y = \lambda f.\left(\lambda \mathrm{x}.\, f\,(x\; x)\right)\left(\lambda x.\, f\,(x\; x)\right)$$

Proprietà fondamentale del combinatore Y

$$Y\; F \equiv_{\beta}\; F\,(Y\; F)$$

Il combinatore Y

$$Y = \lambda f.\big(\lambda \mathrm{x}. f\,(x\ x)\big)\big(\lambda x. f\,(x\ x)\big)$$

$$\begin{aligned} YF \quad &= \quad \lambda f.\big(\lambda \mathrm{x}. f\,(x\ x)\big)\big(\lambda x. f\,(x\ x)\big)F \\ &\to \quad \big(\lambda \mathrm{x}. F(x\ x)\big)\big(\lambda x. F(x\ x)\big) \\ &\to \quad F\ (\big(\lambda \mathrm{x}. F(x\ x)\big)\big(\lambda x. F(x\ x)\big)) \end{aligned}$$

Operatore Y

$$Y = \lambda f.\big(\lambda \mathrm{x}.f(x\ x)\big)\big(\lambda x.f(x\ x)\big)$$

$$\begin{aligned} YF \quad &= \quad \lambda f.\big(\lambda \mathrm{x}.f(x\ x)\big)\big(\lambda x.f(x\ x)\big)F \\ &\to \quad \big(\lambda \mathrm{x}.F(x\ x)\big)\big(\lambda x.F(x\ x)\big) \\ &\to \quad \boldsymbol{F\ (\big(\lambda x.F(x\ x)\big)\big(\lambda x.F(x\ x)\big))} \end{aligned}$$

$$\begin{aligned} \boldsymbol{F\ (\big(\lambda x.F(x\ x)\big)\big(\lambda x.F(x\ x)\big))} \quad &\leftarrow \quad F(\lambda f.\big(\lambda \mathrm{x}.f(x\ x)\big)\big(\lambda x.f(x\ x)\big)F) \\ &= \quad F(Y\ F) \end{aligned}$$

Operatore Y

$$Y = \lambda f.\big(\lambda \mathrm{x}.\, f(x\ x)\big)\big(\lambda x.\, f(x\ x)\big)$$

$$
\begin{aligned}
YF \quad &= \quad \lambda f.\big(\lambda \mathrm{x}.\, f(x\ x)\big)\big(\lambda x.\, f(x\ x)\big)F \\
&\rightarrow \quad \big(\lambda \mathrm{x}.\, F(x\ x)\big)\big(\lambda x.\, F(x\ x)\big) \\
&\rightarrow \quad F\ (\big(\lambda \mathrm{x}.\, F(x\ x)\big)\big(\lambda x.\, F(x\ x)\big)) \\
&\leftarrow \quad F(\lambda f.\big(\lambda \mathrm{x}.\, f(x\ x)\big)\big(\lambda x.\, f(x\ x)\big)F) \\
&= \quad F(Y\ F)
\end{aligned}
$$

Nota: non abbiamo mostrato che $YF \implies F(Y\ F)$ il penultimo passo va nella direzione opposta.
Si può dimostrarlo con la teoria più generale della beta equivalenza

Definizioni ricorsive

Problema: definire una funzione ricorsiva

F = <corpo della funzione contenente F>

Primo passo:

Definire G = λ f. <espressione che contiene f>

Secondo passo:

Definire F = Y G

$$
\begin{aligned}
F \quad &= \quad YG \\
&\equiv_\beta \quad G(\,YG) \\
&\equiv_\beta \quad < espressione\ con\ (Y\ G) > \\
&\equiv_\beta \quad < espressione\ con\ \big(espressione\ con\ (Y\ G)\big) > \\
&\equiv_\beta \quad \ldots.
\end{aligned}
$$

Y combinator in Javascript

```javascript
const Y = f => (x => x(x))(x => f(y => x(x)(y)));

const factorial = f => (x => (x === 1 ? 1 : x * f(x - 1)));

const YFactorial = Y(factorial)(10);
```

Costanti

- Abbiamo fatto vedere come nel lambda calcolo sia possibile definire funzioni, funzioni ricorsive e funzione di ordine superiore.
- Mostriamo come sia possibile codificare nel lambda calcolo valori di verità (Boolean Values) e le operazioni significative per operare su tali valori

Booleans

$$True = \lambda t. \lambda f. t$$
$$False = \lambda t. \lambda f. f$$

Condizionale

$$IF \quad = \quad \lambda c.\lambda them.\lambda else\ .c\ then\ else$$

$$
\begin{aligned}
IF\ True\ e_1 e_2 &= (\lambda l.\lambda m.\lambda n.\, l\ m\ n) True\ e_1 e_2 \\
&\to (\lambda m.\lambda n. True\ m\ n)\ e_1 e_2 \\
&\to (\lambda n. True\ e_1\ n) e_2 \\
&\to True\ e_1 e_2 \\
&= (\lambda t.\lambda f.\, t\ ) e_1 e_2 \\
&\to (\lambda f.\, e_1) e_2 \\
&\to e_1
\end{aligned}
$$

Costanti numeriche: Church Numerals

$$
\begin{aligned}
C_0 &= \lambda z.\, \lambda s.\, z \\
C_1 &= \lambda z.\, \lambda s.\, s\, z \\
C_2 &= \lambda z.\, \lambda s.\, s\,(s\, z) \\
&\vdots \\
C_n &= \lambda z.\, \lambda s.\, \underbrace{s\,(s\,(\ldots(s}_{n \text{ times}}\, z))\ldots)
\end{aligned}
$$

Il naturale *n* è codificato dal numerale C_n funzione con due argomenti

z (lo zero) e s (la funzione successore)

Il numerale C_n applica n-volte la funzione s all'argomento z

Funzioni su naturali

$$\boldsymbol{Plus = \lambda m.\lambda n.\lambda z.\lambda s.m\ (n\ z\ s)s}$$

$$\boldsymbol{Times = \lambda m.\lambda n.m\ C_0(Plus\ n)}$$

Static scoping

Il lambda calcolo adotta un meccanismo di scoping statico per definire la visibilità delle variabili

$$\left(\lambda x.\, x\ (\lambda \mathrm{x}.\, x)\right) z$$

L'occorrenza più a destra della variabile x si riferisce alla variabile x introdotta nel secondo lambda.
Applicando alpha conversione si ottiene una versione equivalente

$$\left(\lambda x.\, x\ (\lambda y.\, y)\right) z$$

Dichiarazioni locali

Le dichiarazioni di variabili locali sono codificate tramite lambda astrazione e aplicazione

let x = e1 in e2 = (λx. e2) e1

Esempio

let x= (λy.y) in x x = (λx.x x)(λy.y)

$(\lambda x.x\ x)\ (\lambda y.y) \rightarrow (\lambda x.x\ x)\ (\lambda y.y) \rightarrow (\lambda y.y)\ (\lambda y.y) \rightarrow (\lambda y.y)$

Strutture dati

Pair(a,b) = λ x . IF x a b

Fst = λ f. f True

Snd = λ f. f False

Fst (a,b) = (λf.f true) (λx.IF x a b)

→ (λx.IF x a b) true

⇒ a

Call by value

La regola di esecuzione Call-by-value (CBV) sassicura che le funzioni siano chiamate solo su valori.

Data l’applicazione (λx. e1) e2 , la regola CBV assicura che e2 sia un valore prima di invocare la funzione.

Nel lambda calcolo, ogni astrazione è un valore. In un lambda calcolo applicato con interi e operazioni aritmetiche, i valori includono anche gli interi.

Riduzione call by value

$$(\lambda x. e_1)v \rightarrow e_1\{x := v\}$$

$$\frac{e_1 \rightarrow e'}{e_1 e_2 \rightarrow e' e_2}$$

$$\frac{e_2 \rightarrow e'}{v\ e_2 \rightarrow\ e_1 e'}$$

Riduzione call-by-name (rivisitata)

$$(\lambda x. e_1)e_2 \rightarrow e_1\{x \coloneqq v\}$$

$$\frac{e_1 \rightarrow e'}{e_1 e_2 \rightarrow e' e_2}$$

Idea: applicare la funzione il prima possibile

Call-by-value: esempio

$$
\begin{aligned}
(\lambda x.\, \lambda y.\, y\ x)\ (5+2)\ \lambda x.\, x+1 \;&\longrightarrow (\lambda x.\, \lambda y.\, y\ x)\ 7\ \lambda x.\, x+1 \\
&\longrightarrow (\lambda y.\, y\ 7)\ \lambda x.\, x+1 \\
&\longrightarrow (\lambda x.\, x+1)\ 7 \\
&\longrightarrow 7+1 \\
&\longrightarrow 8
\end{aligned}
$$

Call-by-Name: esempio

$$
\begin{aligned}
(\lambda x.\, \lambda y.\, y\; x)\; (5+2)\; \lambda x.\, x+1 \;&\longrightarrow (\lambda y.\, y\; (5+2))\; \lambda x.\, x+1 \\
&\longrightarrow (\lambda x.\, x+1)\; (5+2) \\
&\longrightarrow (5+2)+1 \\
&\longrightarrow 7+1 \\
&\longrightarrow 8
\end{aligned}
$$

Discussione

Supponendo che la valutazione di una lambda espressione termini sempre non importa quale ordine di valutazione si sceglie, si ottiene la stessa risposta

Tuttavia, in caso di non terminazione, il comportamento osservabile differisce tra call-by-value e call-by-name

Call by value vs Call by name

Analizziamo il comportamento di **IF True True ($\Omega\ \Omega$)**

Call by Name

IF True True ($\Omega\ \Omega$) $\Longrightarrow$ True

Call by value

IF True True ($\Omega\ \Omega$)

$\Longrightarrow$ IF True True ($\Omega\ \Omega$) $\Longrightarrow$

$\Longrightarrow$ IF True True ($\Omega\ \Omega$)

$\Longrightarrow$ IF True True ($\Omega\ \Omega$) $\Longrightarrow$ ....

Call by Value vs Call by name

IF True True (Ω Ω)

Call by Name

IF True True (Ω Ω) $\Longrightarrow$ True

TERMINAZIONE

Call by value

IF True True (Ω Ω)

$\Longrightarrow$ IF True True (Ω Ω) $\Longrightarrow$

$\Longrightarrow$ IF True True (Ω Ω)

$\Longrightarrow$ IF True True (Ω Ω) $\Longrightarrow$ ....

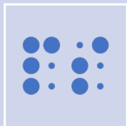
NON TERMINAZIONE

Discussione

Il lambda calcolo è Turing completo

Può rappresentare praticamente qualsiasi costrutto dei linguaggi di programmazione.

Utilizzando codifiche intelligenti

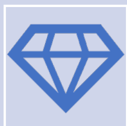


Ma i programmi sarebbero

Piuttosto lenti (migliaia di chiamate a funzioni)

Centinaia di righe di codice)

Difficile da capire



In pratica: usiamo linguaggi più ricchi che includono le opportune primitive linguistiche

La necessità di avere tipi

Nel lambda caklcolo tutti I valori sono codificati tramite le funzioni

- True = λ t. λf . t $=_\alpha$ λ x. λ y. x
- Zero = λ z. λs. z $=_\alpha$ λ x. λ y. x

Pertanto possiamo facilmente usare valori nei contesti sbagliati

- IF 0 e1 e2
- True 0

La stessa cosa accade nei linguaggi di basso livello (assembler)

- Una istruzioni è una parola (di macchina): un mucchio di bit
- Tutte le operazioni operano su parole

Lambda calcolo tipato

il nostro prossimo passo