7/10/2019
testo: Martini (gratis, sintetico)
Asperti-Ciabattoni (a pagamento, facoltativo, per chiarimenti)
formalizzazione del ragionamento matematico

un ragionamento può avere affermazioni certe o ipotetiche

Sistema formale: rappresentazione matematica di un modo di ragionare (es. ragionamenti di medicina, partono da delle conoscenze)
Il sistema formale parte da delle affermazioni/judgments/formule ben formate (fbf), che sono **stringhe di caratteri**.
L'insieme di caratteri usati in un sistema formale si indica con sigma (Σ).
es. CL = {k, s, (,), =}
ESSERE PIU' PRECISO POSSIBILE E DISTINGUERE SINTASSI DA SEMANTICA (rispettivamente rappresentazione e significato) **anche se uno non preclude l'altro**. Mai essere ambigui.
Definire poi un linguaggio.
es W $\in$ Σ^* = {p,e,r}, W = {perepe, perepeperepe, perepeperepeperepe…}
L'insieme delle affermazioni certe si definisce **Ax (assiomi del sistema formale). E' un sottoinsieme dell'insieme dei giudizi/fbf.**
Prendendo in esempio un insieme

Alberto = {P=Q | P,Q $\in$ τ}
stringa P seguita da = seguita da stringa Q dove P e Q $\in$ τ
k $\in$ τ
s $\in$ τ
(MN) $\in$ τ
ks $\in$ τ, (ks)(ks) $\in$ τ, (ks)(ks)k $\in$ τ, [(ss)s] $\in$ τ e così via…

(kP)Q = P **è un assioma**. Gli assiomi sono tutte le fbf/giudizi che hanno struttura elemento/stringa $\in$ τ seguito da = seguito da elemento/stringa $\in$ τ (formalmente {P=Q | P,Q $\in$ τ}).

Passi deduttivi (giudizio1→ giudizio2=giudizio), (MP) **modus ponens**
es.
Se A → (allora) B, affermo B → → → Se ho affermato A e A → B, posso affermare B.
Se io posso affermare perepe, posso affermare perepeperepeperepe.

Definendo un sistema formale CL, dobbiamo stabilire le regole di inferenza.
Regole di inferenza (relazioni, premesse)
es. (linguaggio W)
P=Q / Q=P
P=Q Q=R / P=R (regola T R A N S) (relazione ternaria, sottoinsieme di W3, cioè W x W x W)
(P=Q, Q=R, P=R) è una regola T R A N S $\in$ W x W x W (W^3**)**
…
1. (((sk)k)k) = ((kk)(kk)) → istanza dell'assioma s (Axs)
2. ((kk)(kk)) = k → istanza dell'assioma k (Axk)
3. (((sk)k)k)) = k → T R A N S (1, 2)
4. (((sk)k)k)) = ((kk)(kk)) → ipotesi (Hyp)

1. ((kk)k) = k
2. (kk)k = k
3. (kk)k = k
4. (kk)k = k

Meta-affermiamo che: $\vdash_{CL}$ ((sk)k)k) = k (TEOREMA)
La deduzione è corretta solo se ogni affermazione è un assioma, o se segue le regole di inferenza.
(B) $\vdash_{CL}$ **A** (conclusione A possibile solo grazie all'utilizzo di una ipotesi B)

es.
Sistema formale D
M $\vdash_{D}$ A (posso scrivere una deduzione che ha come conclusione A (che è una fbf) e nella deduzione **POTREI** aver usato una ipotesi/affermazione nell'insieme delle fbf M)

Se M $\vdash_{D}$ A e N è sottoinsieme di M (**FINITO**), allora N $\vdash_{D}$ A.

$\{A_1, A_2, A_3, \dots An\}$ $\vdash_{D}$ B dove A1, A2, A3 possono essere ipotesi
M $\vdash_{D}$ A1
M $\vdash_{D}$ A2
M $\vdash_{D}$ An

dunque

M $\vdash_{D}$ B

Una definizione è un modo univoco per identificare un oggetto o insieme di oggetti.

INSIEME DELLE CONSEGUENZE di un insieme di ipotesi Γ (gamma maiuscolo):
Con(Γ) = {A Є W | Γ $\vdash_{D}$ A}

Consistenza dell'insieme Γ
Se è consistente, non ti permette di derivare tutto
Γ è consistente se esiste almeno una formula A da cui non puoi ottenere un teorema
Con(nessuna ipotesi/insieme vuoto) = W
inconsistente se non è consistente
Con(Γ) = W

Sistema formale D
Insieme di fbf Γ
Γ è una teoria in D quando le affermazioni/ipotesi di Γ ci fanno dedurre conclusioni sempre contenute in Γ.
Con(Γ) = Γ (teoria)
Con(nessuna ipotesi) = { A Є W | $\vdash_{D}$ A } (teoria pura)

DEFINIZIONE ESPLICITA
((sk)k) **Є tau**
I = ((sk)k) [I è il nome dell'elemento, definito esplicitamente]

es. 103
dimostrazione non specificata in CL di {k=sk} |-CL P=Q

R=k
R'=sk
M=kP → kPQ →
N=PQ → (PQ)
R'N=skP
N=P

1. k=sk (ipotesi)
2. kP=kP (refl)
3. kP=skP (cong2)(1.2.)
4. kPQ=kPQ (refl)
5. kPQ=skPQ (congr)(3.4.)
6. kPQ=P (Axk)
7. skPQ=kPQ (symm) (5.)
8. skPQ=P (trans)(7.6)
9. skPQ=kQ(PQ) (Axs)
10. kQ(PQ)=Q (Axk)
11. skPQ=Q (trans)(9.10)
12. P=skPQ (symm) (8.)
13. P=Q (trans)(11.12)

assiomi

		R=R' RM=RN
--------- (Axk)	----------(Axs)	----------(cong2)
kMN=M	sMNR=MR(NR)	RM=R'N

M=N	M=N N=R		
------ (symm)	----------(trans)	--------(refl)	
N=M	M=R	M=M	

35.3
S = {a, b}
---------- (Ax)
aaa

w
---------- (R) (w sarebbe a^n)
waa

b
---------- (R')
baa

R' è derivabile se {b} |-D baa
baa non appartiene a nessuna regola di inferenza, non è un assioma (non è aaaaaaaaa…) né una ipotesi (b =/= baa), quindi non è derivabile

R’ è ammissibile perché all’interno di un sistema formale D+R’, alfa si può derivare direttamente dal sistema D perché la regola R’ non si può utilizzare data la premessa impossibile b, dunque non si conta e si elide (?). Quindi |-D+R’ alfa = |-D alfa
35.4
Il sistema formale D è consistente perché esiste almeno una formula alfa (ad esempio baa) che non è derivabile (dall’ipotesi b), mentre ad esempio aaa si può derivare dall’ipotesi a.
100
Dimostrare in CL che (ksk)(kkk)=sk è un teorema.
((kP)Q)=P (Axk)
1. (ksk) = s (Axk)
2. (kkk) = k (Axk)
3. (ksk)(kkk)=(ksk)(kkk) (assioma riflessivo)
4. (ksk)(kkk)=s(kkk) (cong)(1.3)
5. (ksk)(kkk)=sk (cong)(4.2)

esercizio di tau
soluzione barbanera
1. {k=k, k=s, s=s, s=k} Є W
2. se P=Q, R=T Є W, allora (PR)=(QT) Є W, se P=Q Є W allora (PR)=Q Є W e P=(QR) Є W
3. nient’altro

S = {k, s, (,), =}
W = { P=Q | P, Q appartenente a tau} dove tau è l'insieme dei termini così definito:
1. k, s appartenenti a tau
2. se P, Q appartenenti a tau allora (PQ) appartiene a tau; (P e Q sono delle stringhe)
3. nient'altro è un termine;
Definire l’insieme W senza ricorrere a tau

$$W = \{P=Q \mid P = (k^n s^m)^e (k^o s^p)^f \wedge Q = (k^a s^b)^g (k^c s^d)^h \wedge (PQ) = ((k^n s^m)^e (k^o s^p)^f)((k^a s^b)^g (k^c s^d)^h),\ n,m,o,p,a,b,c,d,e,f,g,h >= 0 \}$$

14/10/2019

correttezza e completezza: implicano la nozione di semantica
semantica di un sistema formale: dare un significato alle fbf, definire un concetto di verità/validità che mi permetta di dire, presa una fbf, se è vera o falsa
(1+2)(3+4)=(5+5)-1 in questo caso è vera se il primo termine è identico al secondo,ma non è così.
Correttezza: tutto ciò che derivi è valido/vero
completezza: tutto ciò che è valido/vero è derivabile

Regole derivabili
Una regola R è derivabile se la regola ha la forma alfa1 … alfa n
/ conclusione beta e se è derivata a partire dall’insieme delle premesse/fbf alfa1...2...alfa n, senza usare la regola stessa (**una regola derivabile è inutile**)
4=4 3+3=6
---------- ®
5+3=8

Regole ammissibili
Se si ha un teorema in |-Du{R} alfa, alfa è un teorema anche se {R} non è compreso in D.

Se una regola è derivabile, è anche ammissibile. Il contrario non è sempre vero.

21/10/19

Un sistema inconsistente non è sempre corretto rispetto a una qualsiasi semantica, ma completo
Un sistema corretto e completo rispetto a una qualsiasi semantica può essere inconsistente.

CALCOLO PROPOSIZIONALE
Sistema formale P0
S = {a, b, c, … z … inf … (,), →, not} variabili proposizionali/proposizioni base
W
p (variabile proposizionale) appartiene a W (ogni fbf è una variabile proposizionale)
Il prato è blu. (affermazione falsa e non valida nel nostro mondo, ma forse vera e valida altrove)
se α **Є** W, allora ¬α **Є** W
se α, β Є W, allora α→ β Є W
((¬p)→r)→(¬p)
assiomi:
α→(β→α) [Ak]
(α→(β→ɣ))→((α→β)→(α→ɣ)) [AS]

regola di inferenza R = {MP} dove MP (modus ponens) è $\dfrac{\alpha \quad \alpha \rightarrow \beta}{\beta}$

teorema di deduzione di Herbrand
se in P0 si può derivare da un insieme gamma di fbf una implicazione alfa→beta, allora dall'insieme gamma + la formula alfa si può derivare beta.
Si dimostra per **induzione**.
se P0 P(m) → P(m+1)
---
allora valida per ogni m di P(m)

oppure **induzione completa**

(matematica)

se P(0) per ogni n vale ((per ogni m<n tale che P(m)) → P(n))
---
allora per ogni n vale P(n)

(logica)

se P(0) per ogni n vale ((per ogni m<n tale che P(m)) → P(n))
---
allora per ogni n vale la seguente: **(se beta è derivabile dalle ipotesi Γ o α con una deduzione lunga n, allora Γ |- α→β)**

28/10/19

se P è corretto rispetto ad una semantica allora se |---P alfa con una derivazione lunga n, nella derivazione sono usati solo assiomi validi e solo regole che permettono di ottenere conclusioni valide se le premesse sono valide.
Se la derivazione è lunga 1, essa è un assioma

ex falso quad libet = da una contraddizione usata come ipotesi puoi dimostrare qualsiasi cosa

alfa, notalfa |-- beta
alfa |-- (notalfa→beta)
|-- alfa→(notalfa → beta)

esercizio:
Usare una generalizzazione della proposizione 2.3 che permette di ottenere la proposizione 3.4

Un insieme gamma di ipotesi in p0 è contraddittorio se e solo se in p0 riesco a derivare (da gamma) una formula alfa e la sua negazione
Se gamma = insieme vuoto, dire che l'insieme gamma vuoto è contraddittorio equivale a dire che p0 è inconsistente.

PROLOG (linguaggio di programmazione usato soprattutto in IA)
Un sistema è corretto se e solo se la conclusione alfa è valida se le ipotesi in gamma sono valide
Un sistema è completo se e solo se

Una formula alfa è valida quando è vera in tutti i mondi possibili.
Cosa significa formula "vera"?
Cos'è un mondo possibile? E' una relazione tra una affermazione di base e il suo valore di verità (v/f, 0/1), anche chiamata funzione di assegnamento proposizionale: B: var.prop. → {0,1}

Prendendo in considerazione una var.prop p, preso un mondo possibile B essa è vera se B(p) = 1, falsa se = 0.
B → ((FBF) → {0,1})

B (not alfa) = {0 se B (alfa) = 1
                {1 se B (alfa) = 0

B (alfa→beta) = if B(alfa) = 1 then B(beta) = 1
B (alfa→beta) = {0 se B (alfa) = 1 e B (beta) = 0
                 {1 altrimenti

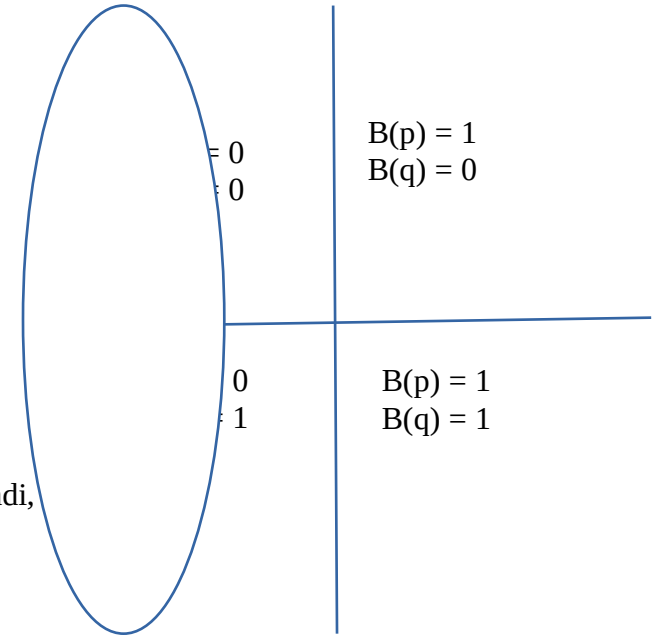
alfa è valida (tautologia) se, per ogni possibile funzione di assegnamento proposizionale, si ha che B (alfa) = 1

4/11/19

Teorema di correttezza

per ogni B, B(a) = 1
è difficile dimostrarlo per ogni B (assegnamento proposizionale)
sia alfa la seguente fbf: (p→q)→¬p
B'(r) = 1
B'(p) = 1
B' (q) = 0
B' (s) = 1
B' (t) = 0
B'' (p) = 1
B'' (q) = 0
B'' (r) = 0
B'' (t) = 0
B'' (s) = 0
Nel mondo B', alfa è vera?

Nel mondo B'', alfa è vera?



Se B(a) non è uguale a 1 in tutti i mondi,
non è una tautologia : |/= a

Per capire se una fbf è una tautologia, si costruisce una tabella di verità:

p	r	(p→r)	(p→r)→r
0	0	1	0
0	1	1	1
1	0	0	1
1	1	1	1

gamma |= a se e solo se per ogni B, qualsiasi sia l'elemento considerato in gamma, se ognuno di essi è B(y) = 1, allora B(a) = 1
DEFINIZIONE DI CORRETTEZZA:
alfa è vera in tutti i mondi dove le formule gamma sono vere
gamma |- a ←→ gamma |= a (conseguenza tautologica)

se B(a) = 1 e B(¬a) = 1, gamma è consistente

se a è una conseguenza tautologica di gamma, alfa è derivabile da gamma

a1, a2, … an |= beta ?

P	q	r	…	A1	a2	a3	…	an	beta
0	0	0		1	1	1		1	1
...	...	...		1	1	0		1	0

sistemi alla Hilbert

sistemi di deduzione naturale
il concetto di regola qui può essere diverso, nonché il concetto di deduzione

$$\frac{a \quad a \to b}{b} \text{ (MP)}$$

esempio di deduzione normale:
1. gamma=delta (ipotesi)
2. delta=gamma (ipotesi)
3 delta (ipotesi)
4 gamma (MP 3,2)
5 delta (MP 4,1)

in deduzione naturale:

$$\frac{\text{delta} \to \text{gamma} \quad \text{delta}}{\text{gamma}} \text{ (MP)} \qquad \frac{\text{delta} \to \text{gamma} \quad \text{gamma}}{\text{delta}} \text{ (MP)}$$

Ax = insieme vuoto

18/11/2019
In un sistema alla hilbert una regola di inferenza è una relazione tra fbf.

Come formalizziamo una regola di inferenza in deduzione naturale?

Affrontiamo un diverso modello computazionale, il lambda-calcolo, diverso da quello utilizzato da linguaggi come il C o il Pascal che è il modello TM (Macchina di Turing).
Una definizione precisa di computazione non esiste. Una definizione più generica sarebbe:

Una computazione è la trasformazione di una informazione da una forma implicita a una esplicita.

TM, lambda-calcolo, S. Post, URM, Ambient-calc, sono tutti formalismi matematici di computazione.
I linguaggi che si servono di questi formalismi sono detti linguaggi di programmazione funzionali e sono ad esempio LISP, PROLOG, Scheme, ML, Haskell…
Nei linguaggi di programmazione funzionali ogni programma è un insieme di definizioni.

Il formalismo delle macchine di Turing è una generalizzazione degli automi a stati finiti: una macchina di Turing è formalizzata infatti come una serie di quintuple.
Tutti i linguaggi di programmazione basati sul modello computazionale delle macchine di Turing si definiscono linguaggi imperativi.
Se ci si basa sul modello Turing, un algoritmo è una macchina di Turing. Nel caso del lambda-calcolo, un algoritmo è un lambda-calcolo e così via.

Il mondo fuori dalle macchine di Turing contiene dei concetti legati ai linguaggi formali, ad esempio le espressioni, che differiscono dalle istruzioni in quanto le espressioni si valutano, le istruzioni si eseguono.
Ci sono cose che entrambi i mondi condividono, come le variabili, e altre no, come la ricorsione o valutazione di un'espressione.
La potenza computazionale dei linguaggi che non utilizzano il modello Turing sta tutta nell'uso della ricorsione.

IMPARIAMO A PROGRAMMARE IN HASKELL

È importante conoscere prima la definizione di **ricorsione**.
Una definizione è ricorsiva quando utilizza il concetto stesso che si sta definendo.

Se ad esempio vogliamo scrivere un programma che permette di trovare l'inverso di una stringa, ci basta applicare la sua definizione ricorsiva (1.41 dell'Ausiello) scritta con la sintassi di Haskell.

4:[1,2,3] : → concatenazione
Il risultato è [4, 1, 2, 3]

[1, 2, 3]++[3, 4, 5] ++ → concatenazione di liste (append)
Il risultato è [1,2,3,3,4,5]

"absdr" = ['a', 'b', 's', 'd', 'r'] → Haskell considera le stringhe come liste di caratteri

Costruiamo la funzione inversa:

inv :: String → String	(dichiarazione: nome funzione, operatore :: che indica che è una funzione che a un tipo String associa un altro tipo String)
inv [] = []	(l'inverso di epsilon è epsilon)
inv (a:ys) = (inv ys)++[a]	(l'inverso di a concatenato ys è uguale all'inverso di ys concatenato alla lista contenente il solo elemento a)

Salviamo il file e carichiamolo nell'interfaccia principale del compilatore Haskell.
Scriviamo adesso:
(inv "dfr")++(inv "acz")
Questo ci dà la stringa "rfdzca" che è quello che cercavamo.

Definiamo la funzione che calcola la lunghezza di una stringa.

length :: String → Int

length [] = 0
length (c:cs) = 1 + (length cs)

Salvando il file, caricandolo e usando la funzione:
length "abcdef"
Il risultato è 6.

Se scriviamo :type length, il comando ci restituisce la dichiarazione della funzione (length :: String → Int).
Se omettessimo questa dichiarazione, la funzione restituirebbe la lunghezza di liste di numeri, di booleani, etc.
length [True, False, True, False]
4
Inoltre, se richiedessimo ad Haskell la dichiarazione della funzione, ci risponderebbe:
length :: Num a => [b] → a

Scrivere il programma che preso delta ci restituisce la funzione delta segnato.

25/11/2019

funzione :: Nat,Nat → Nat

funzione (n,m) = (n*(m*m))

fcur = **funzione curryficata (curryfied)**
fcur : N → (N → N)
(fcur (n)) (m)

(fcur(2)) (3) = 18
(fcur(2)) (4) = 32

Se volessi definire una funzione:
g: N → N
g (m) = 3*m^2

oppure
h: N → N
h (m) = 5*m^2

in versione curryficata potrei definire:
g = fcur(3)
h = fcur(5)

L'uguale ci permette di definire funzioni ma in realtà è l'operatore che ci serve per dare un nome ad una espressione. (es. exp = 5*8)

funzione n = 3*n
Ha il nome "funzione"
la funzione: n |→ n*3 è una funzione anonima, cioè non ha un nome.

In sintassi di Haskell,

"funzione n = 3*n" viene interpretato come "funzione = \n → 3*n"

pippo n = (\m → m*n)
giuseppe = pippo 5

funzione fattoriale
fact 0 = 1
fact n = n * ((fact n) - 1))

PROGRAMMAZIONE FUNZIONALE

I linguaggi funzionali hanno come modello matematico di riferimento il lambda-calcolo.

Cosa sono gli elementi essenziali della programmazione funzionale?

- Variabili (in senso matematico, non sono contenitori di valori e non hanno locazione di memoria)
- Valori base (3, 2, 23…)
- Funzioni base (+, -, *…)

- Applicazione delle funzioni base e non
- Operatore di astrazione funzionale (UNARIA) (\n → …) (partendo da una espressione, rappresento una funzione)
- Assegnamento di un nome alle espressioni (per scrivere i programmi ricorsivi)

Le caratteristiche in verde sono fondamentali, quelle in rosso si possono definire in funzione delle altre.

Il lambda-calcolo è un formalismo matematico in cui definisco un insieme di termini (lambda-termini, che rappresentano espressioni e funzioni).
La lambda-computazione è una serie di passi discreta che trasforma una informazione implicita in una esplicita tramite un processo di trasformazione chiamato beta-riduzione.

Un lambda-termine è una stringa di caratteri generata da una grammatica (sarebbe il sigma).
$\Lambda \rightarrow X$
dove X è un carattere non terminale e rappresenta le possibili variabili:
$X \rightarrow x$
$X \rightarrow y$
$X \rightarrow z$
…

Un lambda-termine può anche essere un lambda-termine applicato a un altro lambda-termine.
$\Lambda \rightarrow (\Lambda * \Lambda)$
$\Lambda \rightarrow \lambda X.\Lambda$

$\Lambda :: = X \mid \Lambda*\Lambda \mid \lambda X.\Lambda$

(. sarebbe la freccetta di Haskell)

λz.((x*y)z) è un lambda-termine.

I lambda-termini rappresentano anche algoritmi per il calcolo delle funzioni.

02/12/2019

Beta riduzione

(\n → n*3) 6
diventa
(n*3)[6/n] = 6*3 = 18

Formalizzazione:

Se si ha un lambda-termine ((λx.M)N), la relazione di beta-riduzione è $((\lambda x.M)N) \rightarrow_\beta M[N/x]$.

Bisogna però definire anche il concetto di “sostituzione”.
Per definire formalmente la sostituzione, dobbiamo definire altri concetti senza i quali la definizione di sostituzione sarebbe incomprensibile.

Introduciamo il concetto di α-conversione (alpha-conversione).
λx.x
λy.y
Sembrano distinti, ma non lo sono.
Sono considerati lo stesso termine grazie al concetto di alphaconversione.
Due termini sono alphaconvertibili se sono identici tranne che per il nome degli argomenti, quindi λx.x e λy.y sono alphaconvertibili.

Sei tutto scemo xDDDDDDDDDDDDDDDDDDDDDDD
sei un prezzemolino.suca

Distinguiamo però tra variabili che sono argomenti e variabili che non lo sono.

In un termine, una variabile è legata (bond) quando è utilizzata come argomento.
λx.x

riduzione strutturale sul termine

L'insieme delle variabili legate (BV) di un termine è così definito:
BV (x) cioè un termine = insieme vuoto
BV (PQ) = BV (P) U BV (Q)
BV (λx.P) = BV(P) U {x}

legata = segnaposto
libera = rappresenta un valore

Due termini sono alphaconvertibili se restano identici rinominando alcune variabili.

(λx.x)x
FV = {x}
BV = {x}
Rinominiamo:
(λy.y)x
FV = {x}
BV = {y}

$\lambda x.(xx) \neq_{alpha} \lambda y.(yx)$

Definiamo ora la **sostituzione**:

Quando noi prendiamo un termine P e sostituiamo al posto di x un termine N (P[N/x]) , l'operazione di sostituzione ha senso solo sulle variabili libere, perché una variabile legata x in un termine esempio (λx.x) sarebbe uguale a una variabile y, z, a, b etc… quindi non ha senso

Formalizziamo

x [L/x] = L perché x è una variabile libera
y [L/x] = y perché x non è presente nel lambda-termine
(PQ)[L/x] = P[L/x] Q[L/x]
se (λx.P)[L/x], qualsiasi sia P la funzione avrà x come argomento e quindi sarà una variabile legata, dunque (λx.P)[L/x] = (λx.P)
(λy.P)[L/x] = λy.P[L/x]

Le definizioni precedenti dicono che:
λy.(zy) [(yy)/z] = λy.((yy)y) → qui lo status della variabile y è cambiato da libera a legata

Dunque le definizioni precedenti sono imprecise e si rischia di modificare lo status delle variabili.
In lambda-calcolo le variabili libere sono ben distinte da quelle legate.
Forniamo quindi condizioni di applicabilità per le definizioni precedenti:

(λy.P)[L/x] = λy.P[L/x] con la condizione che FV(L) intersecato BV(λy.P) = insieme vuoto

Per risolvere il problema, usiamo un termine alphaconvertibile dal precedente:
λx.(zx) [(yy)/z] = λx.((yy)x) che è diverso dal precedente

Quindi,
$((\lambda x.M)N) \rightarrow_\beta M[N/x]$
dove:

- ((λx.M)N) si chiama redex
- M[N/x] si chiama contractum

è definita da quanto sopra.

$x(\lambda z.z)y) \rightarrow_\beta M[N/x]$

Se un termine ha un sottotermine che è un redex, ???

Termini che non contengono redex si chiamano in forma normale.

$x(\lambda z.z)y)[y/z] \rightarrow_\beta$ xy

Come faccio a rappresentare 3 come espressione in lambda-calcolo?
λf. λx. f(f(fx))

I numeri naturali rappresentati in questo modo vengono definiti Numerali di Church, identificati con una sopralinea.

$\overline{N} = \lambda fx.f^n x$

La funzione somma è traducibile come: $\lambda nmfx.nf(mfx)$

La funzione moltiplicazione è traducibile come: $\lambda nmf.n(mf)$

Prendere il termine moltiplicazione e applicarlo ai numerali di Church 3 e 2.

I valori booleani sono traducibili come:
True = $\lambda xy.x$
False = $\lambda xy.y$

Continua nel pdf lambda-calculus...

Tutto ciò che si può calcolare in modo intuitivo si può calcolare con il lambda-calcolo, tuttavia questa tesi non è dimostrabile. (tesi Church-Turing)

9/12/2019

Come fa Haskell a restituirti il comportamento della funzione selezionata con :type funzione ?
Con l'algoritmo del tipo principale.

Strategia di riduzione
Identifica su quale sottoespressione devi applicare il primo passo di betariduzione.
Call by name, call by value sono classi di strategie di riduzione. La prima viene utilizzata da Haskell, la seconda da SKIN.
Ognuna ha pro e contro.

M →betariduzione
→betariduzione
…

Strategia leftmost-outermost
Se sei sicuro di poter ridurre in forma normale, con questa strategia ci arrivi sicuramente.

Ci sono anche strategie ottimali, che ti permettono di arrivare alla forma normale nel numero di passi minore possibile.

Lemma di unicità della forma normale:
Se da un termine M raggiungiamo due forme normali N1 e N2, le forme normali sono automaticamente identiche, cioè alpha-convertibili.

Se da un termine M con una strategia ottieni un termine P e, con un'altra strategia da quel termine M ottieni un termine Q, esiste sempre un termine R in cui P e Q confluiscono.

Beta-conversione (diversa da alpha-conversione)
Studio di una relazione tra lambdatermini che formalizza il concetto di uguaglianza di significato.

$M =_B M'$ [M e M' hanno lo stesso significato ma sono termini differenti]

M e M' sono betaconvertibili se e solo se entrambi confluiscono in uno stesso termine P.

In haskell,
fact = \n → if (n==0) then 1 else n*fact(n-1)

Come lo rappresentiamo in lambdacalcolo?

\n.if (iszero n 0)1(mult n **funzione fattoriale???** (sub n 1)) $=_B$?

Si risolve con una beta-espansione, che è il contrario della beta-riduzione:

(\f.\n.if (iszero n 0)1(mult n **f** (sub n 1))) **fact** $=_B$?

g: D → D
Esistono e appartenenti a D tale che g(e) = e che si chiamano punti fissi della funzione.

Chi mi dice se esiste per un termine M un termine P (il punto fisso) tale che MP $=_B$ P ?
Ogni lambdatermine ammette almeno un punto fisso.

In lambda-calcolo esistono gli operatori di punto fisso, che sono operatori che, applicati a un altro termine, mi restituiscono il suo punto fisso.
Ad esempio, theta.
Operatore Θ (theta): (ΘM) $=_B$ M(ΘM)
Operatore Y: (YM) $=_B$ M(YM) [\f. (\x.f(xx)) (\x.f(xx))]
Entrambi ci danno il minimo punto fisso, il punto fisso con minori informazioni possibili.
Ogni lambda-termine è un punto fisso della funzione identità.

quindi:

Θ**(\f.**\n.if (iszero n 0)1(mult n **f** (sub n 1))) $=_B$ **(\f.**\n.if (iszero n 0)1(mult n **f** (sub n 1)))

(Y(\x.x)) $=_B$ Ω

(Θ(\f.\n.(fn)))
Rappresentare il lambda termine della funzione che cicla su ogni input.

08/01/2020

SEMANTICA FORMALE DEI LINGUAGGI DI PROGRAMMAZIONE

Un linguaggio di programmazione è un linguaggio formale e ha una sua sintassi.
I linguaggi di programmazione solitamente si descrivono attraverso grammatiche.

Come fa il compilatore a dirci se la stringa/programma inserita è corretta?
Prova a generare quella stringa usando la grammatica del linguaggio di programmazione, se è generabile è corretto, altrimenti no.

Un linguaggio di programmazione non è determinato solo dalla sua sintassi ma anche dalla semantica.

La sintassi è un sottoinsieme di stringhe ed è descritta dalla grammatica, la semantica è il significato, cioè il processo computazionale che esse effettuano.

Conoscere un linguaggio di programmazione significa descriverne la sintassi e la semantica in modo preciso.

Un compilatore controlla prima la sintassi, dopo lo traduce in un linguaggio macchina.

Ci sono metodi di descrizione della semantica:

- denotazionali: i metodi più matematici per la descrizione della semantica
- assiomatici: fanno uso del concetto di sistema formale, sono metodi astratti in cui i passi computazionali che esegue un programma quando viene applicato ad un input sono descritti in modo implicito (es. di sistema formale: logica di Hoare, Separation Logic)
- operazionali: semantica più vicina all'implementazione, in cui si descrive in modo formale come avviene la computazione rappresentata dai programmi (es. SOS = semantica operazionale strutturata / structured operational semantics).

Non diremo cos'è una SOS ma daremo un esempio, descrivendo la semantica di un linguaggio di programmazione attraverso la descrizione di un sistema formale (coi relativi assiomi e regole d'inferenza).

Descriviamo il linguaggio WHILE usando la SOS.
La sua grammatica è in BNF (Bakus Normal Form), i simboli tra <> sono simboli non terminali, i simboli in grassetto sono simboli terminali.

Aggiungiamo <digit> → 0 | 1 | 2 … | 9
Stringa corretta: **begin X**1 := **succ** (**X**21); **while X**1 =/= **X**7 **do X**8 := **0 end**

Sistema formale while:

Le formule ben formate hanno forma:
$a^{\rightarrow} P \downarrow b^{\rightarrow}$

↓ è il termine della computazione, quello che c'è dopo è il prodotto della computazione
a e b sono vettori di variabili, P un programma
a0, a1, a2… a_n
b0, b1, b2… b_n

Il programma P, "inizializzato" con i valori a, produce i valori b.

Vedi pdf per maggiori informazioni.

METODI ASSIOMATICI

In quelli operazionali ci si riferisce al contenuto delle variabili, in quelli assiomatici ad affermazioni relative a variabili.

Le formule ben formate hanno forma:
{A} P {B}

15/01/2020

Calcolo dei predicati
E' una estensione della Logica Proposizionale, da cui differisce per la definizione di mondo, che qui viene visto come mondo popolato da individui in cui, tra questi, esistono delle relazioni.
Per costruire le formule ben formate, usiamo i simboli in un insieme S, che come elementi ha soprattutto →, not, i quantificatori (per ogni, esiste, esiste almeno), le parentesi ecc.

Tuttavia, il quantificatore "esiste" non è contemplato dal Martini in quanto si può esprimere con una combinazione di not e → ($\exists x.\alpha = \neg\forall x. \neg\alpha$)

Insieme ai simboli S {→, ¬, (,), ∀} consideriamo anche l'insieme Σ (chiamato anche segnatura), formato da due sottoinsiemi: i simboli di funzione e i simboli di predicato.
Tra i simboli di funzione troviamo quelli che rappresentano individui, mentre i simboli di predicato rappresentano relazioni. Esempio:
Σ = ({m, c, g}, {Q, R}) dove {m, c, g} = isf e {Q, R} = isp
Quindi, in questo caso:
S = {→, ¬, (,), ∀} U isf U isp

Tuttavia, dato che i Σ possono essere diversi, si parla di più logiche del predicato, e quando si parla di una nello specifico ci si deve riferire obbligatoriamente al Σ.

Le formule di base da cui partire per costruire le fbf nel calcolo dei predicati sono più complesse rispetto alla logica proposizionale.

Le stringhe con le quali si denotano gli individui fanno parte del cosiddetto insieme dei termini.
Ai simboli di funzione assegniamo la rispettiva arietà (cioè il numero degli argomenti della funzione). Ad esempio, m^1 c^2 g^0 (g^0 è una funzione con 0 argomenti, quindi è una costante).
Se ho una funzione di arietà 0 tra i simboli di funzione, il simbolo di quella funzione costituisce esso stesso un termine.
Esempi di termini sono: m(g), g, c(m(g), g).

Per esprimere relazioni tra individui, vanno utilizzati anche i simboli di predicato (anch'essi provvisti di arietà, ad esempio Q^3 R^1).

Dato che una relazione intercorre tra due o più elementi, un simbolo di predicato di arietà 1 non è una relazione ma una proprietà.
Per costruire una fbf, prendo un simbolo di relazione e lo applico a un gruppo di n termini (n è l'arietà del simbolo) che denotano altrettanti individui.
Una fbf più complessa potrebbe essere:
∀x: ∀y. (R (c(x,y)) → Q (x, c(x,y), c(m(g), g)

Una fbf è vera quando è valida in tutti i mondi possibili.
Formalizziamo il concetto di mondo possibile o struttura.
Una struttura A^{Σ} (oppure $|A^{\Sigma}|$) è una tripla <A, F, P> dove A è un insieme di individui, F una associazione che associa ad ogni simbolo di funzione una funzione effettiva, P un'altra che associa a ogni simbolo di predicato una relazione.

Come facciamo a capire se una fbf è vera in un mondo possibile/struttura?
Presa [[α]] A^{Σ} (funzione semantica), la fbf è vera o è falsa?

[[Q (g, c(g, g), c(c(g, g), g))]] $^{A\Sigma}$ è falsa
[[R(g)]]$^{A\Sigma}$ è falsa

[[a→b]] $^{A\Sigma}$ è {falsa quando [[a]] $^{A\Sigma}$ = 1, [[b]] $^{A\Sigma}$ = 0
{vera altrimenti

[[∀x.α]] $^{A\Sigma}$ è {vera se [[a]] $^{A\Sigma}$ = 1 per ogni elemento di A^{Σ}
{falsa altrimenti

Ro: Var → A^{Σ} è una funzione ambiente che contiene le associazioni degli individui ai termini (ad esempio x = io, m = la mamma, s = Salvini), ed è necessaria per stabilire il valore di verità di una fbf.

ro_y^a = ambiente dove ad y viene associato a

[[∀x.α]]$_{ro}$ $^{A\Sigma}$ è {vera se [[a]] $_{ro_x^a}$ $^{A\Sigma}$ = 1 per ogni a appartenente a A^{Σ}
{falsa altrimenti

[[∀x.R(x)]]$_{ro}$ $^{A\Sigma}$ è {vera se per ogni a, [[R(x)]] $_{ro_x^a}$ $^{A\Sigma}$ = 1
{falsa altrimenti

Anche nel calcolo dei predicati si distinguono le variabili legate e non libere ed è valido anche il concetto di alpha-conversione.
La quantificazione ∀ ha la funzione del lambda nel lambda-calcolo: lega le variabili.

Vedere pdf deduzione naturale (sezione logica del primo ordine)

N.B. NOZIONE DI VERITA ASSOLUTA: " W SALVINI CHE DIO LO BENEDICA"

22/01/2020

MACCHINE ASTRATTE

Una macchina astratta è un modo per rappresentare in modo astratto sistemi computazionali esistenti. E' un insieme di strutture dati e algoritmi in grado di memorizzare ed eseguire programmi.
Esistono macchine astratte imperative e macchine astratte logiche/funzionali.
Sono formate da diverse componenti:

- Memoria: composta da strutture dati che permettono di conservare i programmi e i dati su cui lavorano
- Operazioni macchina: solo algoritmi
- Controllo sequenza: algoritmi e strutture dati che permettono alla macchina di eseguire nella sequenza corretta le istruzioni del programma conservato in memoria (solo nelle macchine astratte imperative). Per controllare l'esecuzione in sequenza, la macchina astratta è fornita di un puntatore (program counter) che grazie a un algoritmo viene incrementato di 1 ogni termine istruzione

- Controllo trasferimento dati: algoritmi e strutture dati che sono necessari da recuperare i dati su cui lavorare, dalla memoria
- Gestione della memoria: algoritmi e strutture dati che permettono di organizzare al meglio la componente memoria per ottimizzarne l'efficienza
- **INTERPRETE**: contiene un solo algoritmo chiamato algoritmo di **Fetch-Execute**, che coordina tutte le altre componenti e permette l'esecuzione (sempre astratta) dei programmi. Questo algoritmo si compone di un ciclo che inizia con l'acquisizione dell'istruzione da eseguire, procede con la decodifica di essa, acquisisce gli operandi e seleziona l'operazione macchina da seguire e memorizza poi il risultato (oppure ferma il ciclo).
  L'acquisizione dell'istruzione collabora col controllo sequenza e col controllo trasferimento dati per capire quale istruzione deve eseguire, stessa cosa vale per l'acquisizione degli operandi.

Preso un linguaggio di programmazione possiamo associare ad esso una macchina astratta, e viceversa. Quindi possiamo dire che linguaggio di programmazione e macchina astratta sono la stessa cosa, vista però da due prospettive diverse, in quanto la macchina astratta ha una descrizione più reale e tangibile, sulla quale si può realizzare la sua implementazione (cioè il linguaggio di programmazione), in modo da renderla esistente (cioè utilizzabile per scrivere programmi e ottenere un output).

REALIZZARE MACCHINE ASTRATTE

Facciamo in modo che, presa una macchina astratta, diventi esistente e quindi utilizzabile.
Ci sono tre approcci diversi:

- Realizzazione hardware: la macchina viene resa esistente grazie a delle strutture fisicamente esistenti di qualsiasi tipo, tuttavia quando si parla di macchine astratte nello specifico ci si riferisce a circuiti elettronici e transistor. Qualsiasi macchina astratta si può realizzare con circuiti elettronici indipendentemente dalla complessità di essa. Per problemi di costi di realizzazione e progettazione, nonché di flessibilità, le macchine astratte solitamente sono molto semplici e hanno poca potenza espressiva.
- Realizzazione compilativa: ha bisogno di una macchina astratta già esistente. Consiste nello scrivere un programma "C" (compilatore) scritto nel linguaggio L1 associato alla macchina esistente che ha la caratteristica di restituire un programma scritto in L1 se diamo in input un programma scritto in L2 (il linguaggio associato alla macchina che stiamo realizzando). I due programmi hanno semantica denotazionale equivalente. Il programma L1 viene dato in pasto al compilatore, che restituisce il programma equivalente in L2.
  (esercizio: realizzare WHILE su una macchina C++)
- Realizzazione interpretativa: ha bisogno di una macchina astratta già esistente. Consiste nel prendere le strutture dati di M2 e implementarle con strutture dati manipolabili da programmi in M1 di linguaggio L1. Gli algoritmi della macchina M2 (di linguaggio L2) sono realizzati come programmi in L1.

GERARCHIA DI MACCHINE ASTRATTE

Dal punto di vista teorico un sistema di calcolo è formato da tante macchine astratte a cascata, con alla base una macchina astratta realizzata in hardware. Tutto ciò genera una gerarchia di macchine astratte. Tra l'hardware e la macchina astratta che vogliamo implementare dobbiamo fare in modo che ci sia meno distacco possibile in termini di espressività (semantic gap).
Quando il primo livello successivo all'implementazione hardware è fornito a livello interpretativo, si parla di realizzazione firmware.

Curiosità:
La JVM è una macchina associata ad un linguaggio molto meno espressivo di Java, ma contiene delle caratteristiche che rendono l'implementazione di Java, su questa macchina, molto semplice.