λ-calcolo

Vogliamo studiare le problematiche relative al meccanismo di chiamata di funzione (eg differenze fra binding statico e dinamico) in isolamento rispetto alle problematiche dovute ad altri aspetti (eg lo stato, introdotto per gestire aspetti imperativi).

Vogliamo quindi individuare un linguaggio quanto più semplice possibile in cui siano presenti solo meccanismi per

- Astrazione, cioè definizione di funzione
- Applicazione, cioè uso di funzioni

Il linguaggio più semplice possibile è il λ-calcolo, introdotto negli anni ’30 e ancora oggetto di studio.

Deve il nome al fatto che l’astrazione si denota in esso con la lettera λ

Sintassi

Fissato un insieme X di simboli per le variabili del linguaggio, i suoi termini sono definiti induttivamente da:

$$\frac{}{x \in L} \quad x \in X$$

Base

le variabili sono espressioni

$$\frac{e \in L}{\lambda x.e \in L} \quad x \in X$$

Astrazione

rappresenta la funzione con parametro x e corpo e

$$\frac{e_1 \in L \qquad e_2 \in L}{(e_1\ e_2) \in L}$$

Applicazione

rappresenta l'applicazione della funzione e_1 all'argomento e_2

Rispetto ai linguaggi visti finora è estremamente semplificato, non solo perché è più conciso, ma soprattutto perché mancano (gli elementi sintattici corrispondenti a) dei concetti, ad esempio non ci sono definizioni, quindi non avremo bisogno di ambienti né di semantica statica

Esercizio proposto: dare una grammatica context free che descriva lo stesso linguaggio

Semantica Operazionale Strutturata

Vediamo un primo esempio di SOS.

L'idea base è di dare una semantica di *riduzione*: cioè due termini rappresentano la stessa cosa, ovvero hanno la stessa semantica, se si riducono entrambi ad uno stesso termine usando delle regole di semplificazione.

Quindi per dare la semantica basta/bisogna dare le regole di semplificazione.

Le regole sono un'astrazione dei passi di computazione di una macchina interprete e per questo si parla di semantica operazionale.

Le regole seguono, come nel caso della semantica denotazionale, la struttura sintattica dei termini e per questo si parla di semantica operazionale strutturale.

Semantica (SOS): β-regola

L'idea chiave è la corrispondenza fra applicazione ed espansione:

$$\frac{}{(\lambda x.e\ a) \Rightarrow e[a/x]}$$

dove e[a/x] deve essere definito (a parte) in modo da corrispondere alla sostituzione di a (parametro attuale) al posto di x (parametro formale) in e (corpo della funzione).

Nel definire la sostituzione bisogna fare attenzione a due problemi

- Vogliamo sostituire solo le occorrenze di x che corrispondono (sono *legate* da) al parametro formale (λx).
  Ad esempio se e= (λy.λx.(x y) x) solo quella rossa, cioè vanno sostituite solo le occorrenze libere.
- Nell'eseguire la sostituzione non vogliamo che variabili libere del parametro attuale siano catturate da qualche occorrenza legante (λx).
  Ad esempio effettuando una sostituzione brutale in λy.(x y) [y/x] si ottiene λy.(y y) in cui la y globale è stata catturata dal λy (questo corrisponderebbe a binding dinamico).

Occorrenze libere, legate e leganti

(riassunto di cose ben note!)

	x Libera	x Legante	x Legata
	Uso di una variabile "avulsa dal contesto". Si riconduce alla regola di introduzione delle variabili come espressioni	Individua una variabile come "buco" di un contesto	Uso di una variabile in un contesto in cui la stessa variabile rappresenta il "buco"
Analisi	ln(x)+3	f(x)=… ∫…dx	f(x)= ln(x)+3
Logica	p(x)	∀x.… ∃x.…	∃x. p(x)
LP	**if** (x){f(2)} **else** {f(7)}	**void** g(**bool** x){…}	**void** g(**bool** x){ **if** (x){f(2)} **else** {f(7)} }
λ-calcolo	(x x)	λx.…	λx. (x x)

Variabili libere

(altre cose ben note)

Vogliamo definire le variabili libere in un'espressione del λ-calcolo, analogamente a quanto si fa in logica, come l'insieme delle variabili per cui c'e' almeno un'occorrenza libera nell'espressione
(in ($x\ \lambda x.x$) la x compare, da sinistra verso destra, come libera, legante e legata)

Lo facciamo per induzione, seguendo le 3 regole che descrivono le espressioni (termini) del λ-calcolo

$$\frac{}{FV(x)=\{x\}}\ x \in X \qquad \frac{FV(e)=A}{FV(\lambda x.e)=A-\{x\}} \qquad \frac{FV(e_1)=A_1 \qquad FV(e_2)=A_2}{FV((e_1\ e_2))=A_1 \cup A_2}$$

Definizione di sostituzione

ogni occorrenza libera va sostituita

$$\frac{}{x[a/x]=a}$$

le altre variabili restano immutate

$$\frac{}{y[a/x]=y} \quad x \neq y$$

l'applicazione è trasparente

$$\frac{e_1[a/x] = e'_1 \qquad e_2[a/x] = e'_2}{(e_1\ e_2)[a/x]=(e'_1\ e'_2)}$$

le occorrenze legate restano immutate

$$\frac{}{\lambda x.e[a/x]= \lambda x.e}$$

le altre variabili legate restano immutate
facendo attenzione a non legare
eventuali variabili libere

$$\frac{e[a/x]= e'}{\lambda y.e[a/x]= \lambda y.e'} \quad x \neq y \quad y \notin FV(a)$$

$\lambda y.(x\ \lambda x.(x\ y))[(t\ w)/x]= \lambda y.((x\ \lambda x.(x\ y))\ [(t\ w)/x]) =$

$\lambda y.\ ((x[(t\ w)/x])\ (\lambda x.(x\ y)[(t\ w)/x])\ )=$

$\lambda y.\ ((t\ w))\ (\lambda x.(x\ y)))$

□-regola

$$\frac{e[a/x]= e'}{\square\, y.e[a/x]= \square\, y.e'}$$ $x \neq y$ $y \square FV(a)$

…e se $y \square FV(a)$, come si fa a sostituire a al posto di x?

$\square\, y.\ (y\ x)[y/x]=?$

Certamente **non** a $\square\, y.\ (y\ y)$ anzi con le regole attuali non si riduce a nessun □-termine.

Questo non è sensato, perché la scelta del nome di una variabile locale, la y, influenza la possibilità di applicazione della funzione $\square\, x.\ \square\, y.\ (y\ x)$, di cui questo termine è il corpo, a y.

Ma i nomi delle variabili locali non dovrebbero essere rilevanti (neppure visibili all'esterno in altri linguaggi).

Bisogna codificare nel nostro formalismo questo fatto, cioè che $\square\, y.(y\ x)$ e $\square\, z.(z\ x)$ sono per noi la stessa cosa (si noti che $\square\, z.(z\ x)[y/x]= \square\, z.(z\ y)$).

Questo può essere fatto o localmente alla definizione di sostituzione

Questa regola rende non deterministico il calcolo di e[a/x] (perché posso scegliere vari nomi nuovi in caso di collisione)

$$\frac{e[w/y]= e'}{\square\, y.e[a/x]= \square\, w.e'[a/x]}$$ $y \square FV(a) \ \square\ x \neq y$ $w \square FV(a) \ \square\ x \neq w$

o, meglio, lasciare la sostituzione così com'è e aggiungere una regola di riscrittura generale (che prende il nome di □-regola)

$$\frac{}{\square\, x.e\ \square\ \square\, y.e[y/x]}\ y \square FV(e)$$

Questa regola introduce catene infinite di riscrittura di □-termini in cui si continuano a ridenominare variabili locali, per cui non si tratta di "vere" semplificazioni

SOS completa

Due regole di calcolo, che catturano il significato di applicazione funzionale:

β-regola che stabilisce che l'applicazione corrisponde alla sostituzione "intelligente"

$$\frac{}{(\lambda x.e\ a) \rightarrow e[a/x]}$$

α-regola che stabilisce che i nomi delle variabili locali sono irrilevanti

$$\frac{}{\lambda x.e \rightarrow \lambda y.e[y/x]}\quad y \notin FV(e)$$

Più regole di chiusura contestuale, che stabiliscono le strategie di valutazione.

Ad esempio la più liberale possibile:

$$\frac{e \rightarrow e'}{\lambda x.e \rightarrow \lambda x.\ e'} \qquad \frac{e_1 \rightarrow e'_1}{(e_1\ e_2) \rightarrow (e'_1\ e_2)} \qquad \frac{e_2 \rightarrow e'_2}{(e_1\ e_2) \rightarrow (e_1\ e'_2)}$$

Utilità del β-calcolo

La sua estrema semplicità permette di mettere bene a fuoco alcune problematiche:

- La definizione di sostituzione decide se il binding è statico (come lo abbiamo fatto) o dinamico (se si toglie la condizione a lato in rosso dell'ultima regola).
- Aggiungendo condizioni alla β-regola se ne può forzare l'applicazione solo ai casi in cui l'argomento sia già stato valutato, per ottenere la semantica di call-by-value (quella standard è la call-by-need)
- Provare la confluenza del sistema dato è (stato fatto, quindi è) fattibile.
- Si capisce bene come sia possibile avere per uno stesso termine due catene di riscrittura, una delle quali finita e l'altra no: (λx.t (λy.(y y) λy.(y y))) si può riscrivere in t usando la β-regola sull'applicazione più esterna, ma può anche essere origine di una riscrittura infinita se si espande l'argomento (λy.(y y) λy.(y y)) usando su di esso la β-regola

Non ha però senso "imparare a programmare" in β-calcolo

Potenza del λ-calcolo

Per essere semplice è semplice, ma ci si può fare qualcosa?
Incredibile ma vero: è un linguaggio universale

Si possono codificare in esso valori base:
n diventa $\lambda x.\lambda y.(\underbrace{(...((x\ x)\ x)...x)}_{\text{n volte}}\ y)$

true diventa $\lambda x.\lambda y.x$ e false diventa $\lambda x.\lambda y.y$

Ad esempio la scelta condizionale diventa $\lambda c.\lambda r_1.\lambda r_2((c\ r_1)\ r_2)$

Si possono descrivere funzioni parziali: $\lambda x.\ (x\ x)$

Si può codificare (in vari modi) un operatore di iterazione (punto fisso):

$\lambda f.(\lambda y.((f\ (y\ y))\ (f\ (y\ y))))$

Usando il currying si possono definire (elementi isomorfi a) funzioni di più variabili

Quindi si presta ad analizzare varie problematiche.

Semantica denotazionale (?)

Per poter dare una semantica denotazionale del λ-calcolo dovremmo anzitutto definire un insieme (dominio) di interpretazione D.

Intuitivamente gli elementi rappresentati dai λ-termini sono funzioni, quindi D deve essere della forma $[D \to_P D]$.

Siccome si possono applicare a se stesse, $D = D$ e siccome il risultato dell'applicazione è a sua volta accettabile come argomento $D = D$.

Perciò $D = [D \to_P D]$, ma questo è possibile?

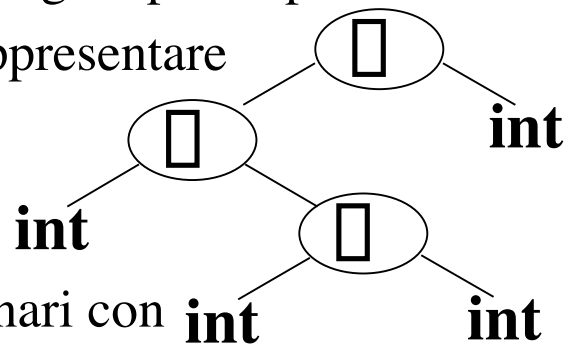
La risposta è affermativa, ma la dimostrazione non è banale.

Il punto cruciale è che si può dimostrare che la funzione Fun che associa un insieme X all'insieme $[X \to_P X]$ ammette un (minimo) punto fisso, cioè esiste un insieme D tale che D = Fun(D) (ed è incluso in ogni punto fisso Y, cioè se Y=Fun(Y), allora $D \subseteq Y$).

Punto fisso ed induzione

Fino ad ora non abbiamo avuto bisogno della teoria del punto fisso, perché tutti i problemi tecnici solitamente risolti con essa (eg semantica di iterazione e ricorsione) li abbiamo risolti usando l'induzione. Cerchiamo di capire perché qui fallisce.
Nel caso di un linguaggio applicativo tipato, come LF, i tipi e la loro semantica sono definiti induttivamente a partire dai tipi base grazie ad una regola per i tipi funzionali.

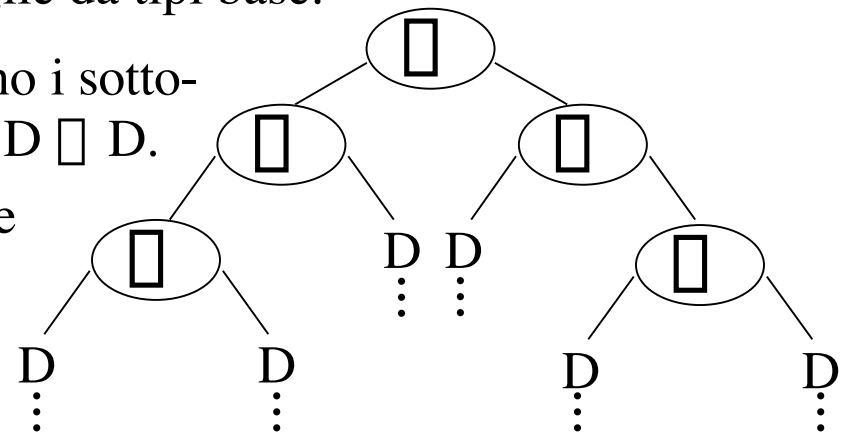
Quindi ad esempio, usando una notazione ad albero per rappresentare i tipi funzionali, (**int** □ (**int** □ **int**)) □ **int** è
Analogamente l'interpretazione semantica avrà lo stesso albero, con sulle foglie gli insiemi corrispondenti



In generale, i tipi funzionali sono rappresentati da alberi binari con tutti i nodi intermedi etichettati da frecce e le foglie da tipi base.

Proviamo la stessa tecnica, espandendo man mano i sotto-alberi da foglie etichettate con D ad alberelli per D □ D.

Si ottiene un albero binario infinito perfettamente bilanciato con tutti i nodi etichettati dalla freccia (e nessuna foglia).



Rispetto alla definizione induttiva di albero binario, per ottenere anche quelli infiniti serve l'analogo di un passaggio al limite, ovvero il calcolo di un punto fisso (che è il limite in un'opportuna teoria)