Linguaggi con Tipi di Ordine Superiori

Aniello Murano
Università degli Studi di Napoli
"Federico II"

Prefazione

- In questa seconda parte del corso parleremo di linguaggi di ordine superiore.
- Si tratta di linguaggi più articolati di IMP in cui sono ammesse anche funzioni come valori possibili.
- In questa lezione considereremo un linguaggio Eager. Di questo linguaggio vedremo:
  - **La sintassi dei termini** del linguaggio (cioè gli oggetti che possiamo costruire a partire da una grammatica data)
  - **Il calcolo del "tipo"** associato ad ogni termine, cioè stabilire il trermine di che tipo è (per esempio il termine 3+5 è di tipo intero).
  - **La semantica operazionale Eager** per il calcolo del significato di un termine del linguaggio considerato.

I tipi nei linguaggi di programmazione

- Un **sistema dei tipi** per un linguaggio di programmazione è un insieme di regole che consentono di dare un tipo ad espressioni, comandi ed altri costrutti del linguaggio.
- **Un linguaggio si dice tipato** se per esso è definito un tale sistema; altrimenti si dice **non tipato**.
- **Il processo che porta alla determinazione di un tipo** per i termini di un linguaggio si chiama **controllo dei tipi** (type checking).
- **Lo scopo di un sistema dei tipi** è quello di evitare che durante l'esecuzione di un programma occorrano errori quali, ad esempio, l'applicazione di funzioni ad argomenti inappropriati o il riferimento illegale della memoria.
- **La definizione di una teoria dei tipi,** ovvero un sistema dei tipi dato sotto forma di regole formali di deduzione, permette, tramite una semantica formale, di dimostrare matematicamente **proprietà del linguaggio**.

Linguaggi con tipi di Ordine Superiore

- Si tratta di linguaggi che ammettono anche funzioni come valori: *dunque, è possibile scrivere una procedura che prende in input una funzione e restituisce una funzione!*
- La progettazione dei linguaggi funzionali è basata su **funzioni matematiche** (a differenza di quelli imperativi, basata sulla architettura di von Neumann degli elaboratori).
- La base teorica dei linguaggi funzionali è rappresentata dal lambda-calcolo.
- il λ-calcolo è stato definito da Alonso Church nel 1933

Alonzo Church (14 giugno, 1903 – 11 agosto, 1995), matematico e logico americano, co-fondatore dell'informatica.

In un suo famoso articolo del 1936 mostra l'esistenza di un "problema indecidibile". Questo risultato precedette il famoso lavoro di Alan Turing sul problema della fermata

Il λ-calcolo

- il λ-calcolo è un formalismo generale per rappresentare funzioni
- il λ-calcolo definisce la stessa classe di funzioni che è definita dalle Macchine di Turing
- da questa uguaglianza la tesi di Church che congettura che questa sia la classe di tutte le funzioni calcolabili
- λx.t è una λ-astrazione definisce una funzione con un parametro "formale" x. Ad esempio λx.x+1 denota la funzione successore.
- In λx.t, λ lega x nel termine t (scope del legame) ed x è legata (bound) in λx.t
- Importante: λx.t è solo un segna-posto e possiamo rinominarla a piacimento, rinominando anche le occorrenze di x in t.
  - Ad esempio, la funzione identità può essere scritta λx.x oppure λy.y

Tecniche di Valutazione

- Esistono due tecniche fondamentali di valutazione per i linguaggi funzionali:
- **Eager**/Call by value: Gli argomenti sono valutati prima della chiamata a funzione
- **Lazy**/Call by name: Gli argomenti sono valutati quando essi vengono utilizzati dalle funzioni
- La scelta Eager/Lazy porta a linguaggi il cui comportamento è riconducibile:
  - nel caso Eager, a quello dei linguaggi ML, LISP e Schema (un dialetto di LISP);
  - nel caso Lazy, a quello di Miranda, Orwell e Haskell.

Valutazione Eager

- La valutazione Eager è la strategia di valutazione maggiormente utilizzata nei linguaggi di programmazione.
- Nella valutazione Eager, una espressione è valutata non appena viene assegnata ad una variabile.
- I vantaggi della valutazione Eager risiedono in un minor consumo di memoria e una maggiore velocità di esecuzione dei programmi.
- Per esempio, si considerino le seguenti istruzioni:
  - `x = 5 + 3 * (1 + 5^2);`
  - `print x;`
  - `print x + 2;`
- In questo caso non solo la valutazione Eager permette di salvare spazio (in quanto conserviamo il valore 83 dell'espressione, invece dell'espressione stessa), ma permette di valutare l'espressione una sola volta e non ogni volta che viene usata.

Valutazione Lazy (prossima lezione)

- La valutazione Lazy (conosciuta anche come valutazione ritardata) consiste nel posticipare la valutazione di una computazione fino a quando (e ogni volta) il risultato della computazione è realmente usato.
- Tra i vantaggi della valutazione Lazy ricordiamo:
  - Aumento delle performance di un programma in seguito alla non valutazione di componenti non necessari alla computazione;
  - Riduzione della possibilità di incontrare errori nella valutazione condizionale di computazioni (l'errore potrebbe essere nella diramazione di un comando "if" che non verrà mai preso);

Linguaggio Eager

- Esistono termini la cui valutazione (Eager) può produrre valori di base, come per esempio i numeri, oppure coppie di valori o funzioni.
- Per tener conto della diversa natura dei valori prodotti dalla valutazione dei termini, si introducono i **tipi** per le espressioni, indicati con τ, e definiti come segue:

$$\tau ::= \text{int} \mid \tau_1 * \tau_2 \mid \tau_1 \rightarrow \tau_2$$

dove

- int → termine valutato con numero
- $\tau_1 * \tau_2$ → termine valutato coppia
- $\tau_1 \rightarrow \tau_2$ → termine valutato funzione

Sintassi dei termini

- Assumendo che a ciascuna variabile **x** di **Var** sia associato un unico tipo, dato da type(x), la sintassi dei termini è data da

$t ::= x \mid$ *names*

– these are bound names, not variable locations

$n \mid t_1 + t_2 \mid t_1 - t_2 \mid t_1 \times t_2 \mid$ *arithmetic*

if t_0 **then** t_1 **else** $t_2 \mid$ *conditional*

– tests if t_0 is zero

$(t_1, t_2) \mid$ *pair formation*

fst$(t) \mid$ **snd**$(t) \mid$ *projection*

$\lambda x.\ t \mid$ *abstraction*

$t_1\ t_2 \mid$ *application*

let $x \Leftarrow t_1$ **in** $t_2 \mid$ *prior evaluation*

rec $y.(\lambda x.\ t)$ *recursive function*

– only allows definition of recursive functions, not pairs

astrazione di un termine t rispetto ad una variabile x

La funzione che, quando applicata ad un valore v, produce t in cui v rimpiazza x

applicazione di una funzione ad un argomento: la funzione t_1 applicata all'argomento t_2

Termini tipabili

- Un termine t è tipabile, cioè $t : \tau$, se corrisponde a un tipo:

  int $\quad \tau_1 * \tau_2$ oppure $\tau_1 \rightarrow \tau_2$

  secondo le seguenti regole (di tipo):
- Variabili
  - $x : \tau$ se type(x) = τ
- Operazioni
  - n : int
  - se t_1 : int & t_2 : int allora t_1 op t_2 : int (dove op è una operazione aritmet.)
- Per i prodotti e le funzioni si veda Winskel a pagina 203
- Let e REC

$$\frac{x : \tau_1 \quad t_1 : \tau_1 \quad t_2 : \tau_2}{\textbf{let } x \Leftarrow t_1 \textbf{ in } t_2 : \tau_2} \text{ [T-let]} \qquad \frac{y : \tau \quad \lambda x.\, t : \tau}{\textbf{rec } y.(\lambda x.\, t) : \tau} \text{ [T-rec]}$$

- Un termine è univocamente tipato se esiste un solo tipo possibile corrispondente (si provi per esercizio che questo vale per i termini considerati)

Esercizio 1

- Verificare se il termine seguente è tipabile. In tale caso, mostrarne il tipo, descrivendo tutte le regole di tipo utilizzate.

  if fst(x) then (snd(x), fst(x)) else x

- Soluzione [Sketch]:
- Dalla regola di tipo dell 'if, segue

$$\frac{\text{fst(x): int} \qquad \text{(snd(x), fst(x))} : \tau \qquad \text{x: } \tau}{\text{if fst(x) then (snd(x), fst(x)) else x: } \tau}$$

- Per la regola di fst(x), segue che x è di tipo coppia con il primo termine intero, dunque $x: \tau \equiv int*\tau_1$ per un opportuno τ_1
- Siccome $(snd(x), fst(x)) : \tau \equiv int*\tau_1$ segue che snd(x): int
- Dunque il termine è di tipo $\tau \equiv int*int$

Esercizio 2

- Provare che $\lambda x.\ \lambda y.\ \lambda z.$ if z then (x y) else (y x) non è tipabile
- Soluzione[Sketch]:
- Tralasciando la valutazione per λ e per l'if (da completare per esercizio), per le funzioni si ha:

$$\frac{x: \tau_1 \to \tau_2 \qquad y: \tau_1}{(x\ y): \tau_2} \qquad\qquad \frac{y: \tau_3 \to \tau_4 \qquad x: \tau_3}{(y\ x): \tau_4}$$

Dalle regole precedenti segue che $\tau_1 \equiv \tau_3 \to \tau_4$

Siccome $\tau_3 \equiv \tau_1 \to \tau_2$, segue che $\tau_1 \equiv \tau_1 \to \tau_2 \to \tau_4$ che è ricorsivo.

Dunque il termine non è tipabile

Esempi

- Calcolo del Fattoriale di n (per n>0):
- rec fact.(λn. (if n-1 then n else n * fact(n-1)))

- Calcolo dell'n-esimo numero di Fibonacci:
- rec fib.(λn. (if n then n else if n-1 then n else fib(n-1) + fib(n-2)))

- Uno sguardo a O'Caml per il numero di Fibonacci:
- let rec fib = fun n -> if n < 2 then n else fib(n-1) + fib(n-2)

- Esercizio: Scrivere il termine per la funzione ricorsiva $f(n)=2^n$

Variabili libere di un termine

- in x+3 la variabile x è libera
- Ricordiamo che λ è un operatore che lega (binds) una variabile (binding operator). Un'occorrenza di una variabile x legata da un operatore λ (cioè che compare nello scope di un λx.) non è libera.
- L'insieme delle variabili libere di un termine t è definito per induzione strutturale su t (vedi Winskel pagina 205 per le regole)
- Un termine è chiuso quando non contiene variabili libere.

Sostituzione

- **La sostituzione** consiste nel rimpiazzo di tutte le occorrenze di un sotto-termine con un altro, all'interno di un terzo termine.
- Indichiamo con

$$[x \rightarrow T_2]T_1 \quad \text{oppure} \quad T_1[T_2 / x]$$

la sostituzione del termine T_2 al posto di ogni occorrenza di variabile libera x all'interno del termine T_1, il quale funge da contesto.

- Un esempio di sostituzione è il seguente:

$$[x \rightarrow t^2]\ \lambda y.\ \log(y + x) = \lambda y.\ \log(y + t^2)$$

Regole di sostituzione (I tentativo)

- Di seguito una definizione ricorsiva dell'algoritmo di sostituzione
  1. $[x \rightarrow t_1]\, x = t_1$;
  2. $[x \rightarrow t_1]\, y = y$; se $x \neq y$
  3. $[x \rightarrow t_1]\, \lambda y.t = \lambda y.([x \rightarrow t_1]\, t)$
- Cosa succede allora se applico la seguente sostituzione:

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

- Ottengo $\lambda x.y$ cioè la variabile legata è diventata libera!!

Regole di sostituzione (II tentativo)

- Di seguito una definizione ricorsiva dell'algoritmo di sostituzione
  1. $[x \rightarrow t_1]\, x = t_1$;
  2. $[x \rightarrow t_1]\, y = y$; se $x \neq y$
  3. Distinguiamo allora i seguenti casi:
     a. $[x \rightarrow t_1]\, \lambda x.t = \lambda x.t$
     b. $[x \rightarrow t_1]\, \lambda y.t = \lambda y.([x \rightarrow t_1]\, t)$ se $x \neq y$
- Cosa succede allora se applico la seguente sostituzione:

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

- Otteniamo $\lambda y.y$ cioè una variabile libera è divenuta legata dopo la sostituzione!! (fenomeno di cattura di variabili). Ciò è dovuto al fatto che la variabile y legata al λ è anche presente tra le variabili libere del termine che t_1 che subentra (cioè $y \in FV(t_1) = FV(y) = \{y\}$.

Regole di sostituzione (definiz. finale)

- Di seguito una definizione ricorsiva dell'algoritmo di sostituzione
  1. $[x \rightarrow t_1]\ x = t_1$;
  2. $[x \rightarrow t_1]\ y = y$; se $x \neq y$
  3. Distinguiamo allora i seguenti casi:
     a. $[x \rightarrow t_1]\ \lambda x.t = \lambda x.t$
     b. $[x \rightarrow t_1]\ \lambda y.t = \lambda y.([x \rightarrow t_1]\ t)$ se $x \neq y$ e **$y \notin FV(t_1)$**
     c. $[x \rightarrow t_1]\ \lambda y.t = \lambda z.([x \rightarrow t_1]\ [y \rightarrow z]t)$ se $x \neq y$ e **$y \in FV(t_1)$**

     Dove z è una nuova variabile non presente né in t, né in t_1

Semantica operazionale eager

- Nella semantica operazionale si utilizzano **forme canoniche** dei termini mentre nella semantica denotazionale si usano i loro **valori**.
- *In generale, un insieme di regole che esprimono un dato concetto sono in **forma canonica** se esse sono nella forma più semplice possibile, senza però perdere di generalità*
- $t \in C^e_\tau$ indica che t è una forma canonica di tipo τ, ed è definita per induzione strutturale su τ nel modo seguente:
- $n \in C^e_{int}$ (i numeri sono forme canoniche)
- $(c_1,c_2) \in C^e_{\tau_1 * \tau_2}$ se $c_1 \in C^e_{\tau_1}$ e $c_2 \in C^e_{\tau_2}$;
- $\lambda x.t \in C^e_{\tau_1 \rightarrow \tau_2}$ se $\lambda x.t : \tau_1 \rightarrow \tau_2$ e $\lambda x.t$ è chiuso
- Dunque le forme canoniche sono particolari termini chiusi
- Definiamo ora le regole di inferenza in grado di ridurre un termine chiuso tipabile t in una forma canonica c, cioè $t \rightarrow^e c$.

Valutazione di operatori e condizioni

$c \rightarrow^{e} c$ for $c \in C_T^e$

$$\frac{t_1 \rightsquigarrow^{e} n_1 \qquad t_2 \rightsquigarrow^{e} n_2}{t_1\ op\ t_2 \rightsquigarrow^{e} c_1}\ [op]$$

$$\frac{t_0 \rightsquigarrow^{e} 0 \qquad t_1 \rightsquigarrow^{e} c_1}{\textbf{if}\ t_0\ \textbf{then}\ t_1\ \textbf{else}\ t_2 \rightsquigarrow^{e} c_1}\ [\text{if-T}]$$

takes zero as true

$$\frac{t_0 \rightsquigarrow^{e} n \quad n \neq 0 \quad t_2 \rightsquigarrow^{e} c_2}{\textbf{if}\ t_0\ \textbf{then}\ t_1\ \textbf{else}\ t_2 \rightsquigarrow^{e} c_2}\ [\text{if-F}]$$

Valutazione delle coppie

- Strict evaluation
  - Both components of pair must be evaluated before projections can be taken

$$\frac{t_1 \rightsquigarrow^{e} c_1 \quad t_2 \rightsquigarrow^{e} c_2}{(t_1, t_2) \rightsquigarrow^{e} (c_1, c_2)}\ [\text{pair}]$$

$$\frac{t \rightsquigarrow^{e} (c_1, c_2)}{\textbf{fst}(t) \rightsquigarrow^{e} c_1}\ [\text{fst}] \qquad \frac{t \rightsquigarrow^{e} (c_1, c_2)}{\textbf{snd}(t) \rightsquigarrow^{e} c_2}\ [\text{snd}]$$

Valutazione di funzioni, let e rec

- Eager Application
  - t_2 is evaluated fully to c_2 before being substituted for x in t'_1
  - t'_1 is also evaluated before being used
- Let evaluates t_1
- Rec returns a canonical form by unfolding once

$$\frac{t_1 \rightsquigarrow^e \lambda x.t'_1 \quad t_2 \rightsquigarrow^e c_2 \quad [x \mapsto c_2]t'_1 \rightsquigarrow^e c}{t_1\ t_2 \rightsquigarrow^e c}\ [app]$$

$$\frac{t_1 \rightsquigarrow^e c_1 \quad [x \mapsto c_1]t_2 \rightsquigarrow^e c}{\textbf{let}\ x \Leftarrow t_1\ \textbf{in}\ t_2 \rightsquigarrow^e c}\ [let]$$

$$\textbf{rec}\ y.(\lambda x.t) \rightsquigarrow^e \lambda x.([y \mapsto \textbf{rec}\ y.(\lambda x.t)]t)\ [rec]$$

Esempi ed esercizi

- Il termine (λx. λy. x+y 5) non è in forma canonica, mentre lo è il termine (λy. 5+y)
- Esercizio: Si derivi la valutazione del termine
  rec f. (λz. if z then 1 else z * f(z-1))

Attenzione

- **Non sempre è possibile scrivere un termine in forma canonica. Quando questo non è possibile, abbiamo una situazione analoga a quella di computazioni non terminanti viste per IMP.**
- Per esempio, il termine t ≡ (**rec** f. (λx. (f x) 5)) non ha forma canonica.
- Il termine t è una applicazione (t_1 t_2).
- Per applicare la regola corrispondente,

$$\frac{t_1 \rightsquigarrow^e \lambda x.t'_1 \quad t_2 \rightsquigarrow^e c_2 \quad [x \mapsto c_2]t'_1 \rightsquigarrow^e c}{t_1\ t_2 \rightsquigarrow^e c}\ [app]$$

siccome t_2 è già in forma canonica, occorre lavorare solo su t_1

- t_1 è un rec e per la regola corrispondente abbiamo:

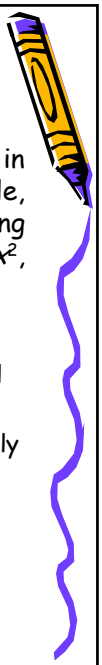
  rec f. (λx. (f x) →^{e} λx.[f →t_1] (f x) ≡ λx. (**rec** f. (λx. (f x) x)

- Ritornando alla regola delle applicazioni, la forma canonica di t è data da [x→ 5] (**rec** f. (λx. (f x) x) ≡ t.
- Si noti come, per le regole di sostituzione, la x nello scope del lambda non viene sostituita (la seconda è invece libera).

La valutazione è deterministica

- Per dimostrare che la valutazione è deterministica occorre provare che per ogni termine se t →^{e}c e t →^{e}c' allora c ≡ c'.
- La precedente proprietà è facilmente dimostrabile per induzione sulle regole di derivazione date (esercizio per gli studenti).

Forme Canoniche



- In mathematics, a **canonical form** is a function that is written in the most standard, conventional, and logical way. For example, polynomials are usually written with the terms in descending powers: it is more usual to write $x^2 + x + 30$ than $x + 30 + x^2$, although the two forms are essentially equivalent.
- A canonical form is required to have two essential properties:
  - Every object under consideration must have exactly one canonical form,
  - Two objects that have the same canonical form must be essentially the same.