Linguaggi di Programmazione

Ivano Salvo

Algoritmi e Programmi
Intro ai Linguaggi Funzionali

Corso di Laurea in Matematica

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **1**, 29 settembre 2020

Lezione 1a:

Introduzione al corso

Programmazione vs Algoritmi

Qual è la differenza tra la **programmazione** e gli **algoritmi**?

Prendete un algoritmo di cui avete dimostrato la correttezza: scrivete il programma e testatelo: usualmente non funziona.

Ma cosa può andare storto? Tipicamente:

- **Operazioni astratte non** implementate correttamente
- **side-effects** e altri problemi legati alla **memoria**
- Anche l'**efficienza** del programma potrebbe essere non coerente con la complessità asintotica calcolata

Esempio: riflettete su quante sottigliezze possano nascondere soprattutto le liste e altre strutture dinamiche.

Esempio (da Informatica Generale)

Esercizio 2 (10 punti) Considerare il problema di verificare se una lista di caratteri è palindroma, cioè se letta da sinistra a destra o da destra a sinistra dà la stessa sequenza di caratteri (come le parole 'non', 'otto', 'radar', 'ingegni', 'onorarono', 'ossesso', ...).

1. (**3 punti**) Considerate il seguente programma (dove la funzione `eqList` verifica se due liste sono uguali e `reverse` restituisce il puntatore alla testa di una lista che contiene gli elementi di `L` in ordine rovesciato):

```
int palindroma(charList L){
    if (eqList(L, reverse(L)) return 1;
       else return 0;
   }
```

   Sotto quali ipotesi sulla funzione `reverse` la funzione dà risultati corretti?

Si intascavano comodamente **3** punti semplicemente dicendo che `reverse` **deve lasciare** `L` **immutata**. Cioè deve essere come `reverseFun` e **creare una nuova lista risultato**.

Assolutamente **imprevedibili i risultati** se L viene modificata.

Computabilità e Pratica della Programmazione

Da un punto di vista **puramente teorico**, bastano modelli di calcolo (e linguaggi di programmazione) estremamente minimali.

A Informatica Generale abbiamo visto il TinyC, ma potremmo pensare a linguaggi ancora più elementari: puntatori (**celle di memoria**) e possibilità di saltare a una certa istruzione sulla base della verifica di una certa condizione (**salto condizionato**)

Ma vogliamo questo?

I linguaggi di programmazione devono fornire **astrazioni sul controllo** (ad esempio: **cicli**, **funzioni** etc.) e **astrazioni sui dati** (definizione dei **tipi di dato**, con operazioni e regole di visibilità)

"**Un linguaggio deve fornire un piccolo numero di metafore uniformi con chiara semantica**"
[Dan Ingalls "**Desing Principles behind Smalltalk**"]

Evoluzione dei Linguaggi di Programmazione 1

- **Linguaggi Macchina**
  - **Istruzioni**: trasferimento di memoria/aritmetiche, **identificate da stringhe di bit**
  - completo controllo del processore e della memoria
  - **Tipo di dato**: celle di memoria (**stringhe di bit!**)
  - **Controllo**: salto (in)condizionato
- **Linguaggi Assembler**
  - Uso di **nomi e label simbolici** per istruzioni e dati (`add %d1 3` invece di `000123 010012 0000011`)
  - 1:1 con istruzioni macchina
  - Uso di label simboliche: **astrazioni utile?** [se aggiungo/tolgo un'istruzione non devo modificare tutti i salti]

Evoluzione dei Linguaggi di Programmazione 2

- **Linguaggi Imperativi**
  - **Istruzioni**: Assegnazioni, cicli
  - **Tipo di dato**: **astrazione** delle celle di memoria: controllo sui tipi
  - Definizioni di **nuovi tipi di dato**.
  - **Controllo**: cicli + **astrazione procedurale/funzionale**.
- **Linguaggi Funzionali**
  - Definizione di **funzioni ricorsive**.
  - Controllo: **riduzione di espressioni**.
  - **Niente memoria**, ma ambiente di valutazione.
  - No alias/side-effects: **ricca teoria algebrica**.
- **Linguaggi Logici**
  - Definizione logica della specifica genera un meccanismo computazionale.
  - Programmare **cosa** invece di programmare **come**.

Riassumendo: Famiglie di Linguaggi di Programmazione

- **Imperativi**: un programma è essenzialmente una **sequenza di comandi** (sequenza, salti, cicli) che modificano la **memoria** di una macchina (**assegnazione**). L'esecuzione è una sequenza di trasformazioni della memoria: **Algol**, **Fortran**, **C**, **Pascal**, ...
- **Orientati agli Oggetti**: il programma consiste nella definizione di **classi** (≈ tipi) da cui creare **oggetti** (≈ dati). L'esecuzione di un programma è costituito dall'interazione di oggetti (**scambio di messaggi**). Gli oggetti possono essere visti come dati o piccoli automi. **Smalltalk**, **Java**, **C++**, …
- **Funzionali**: un programma è una sequenza di definizione di **funzioni ricorsive**. L'esecuzione consiste nella **riduzione di espressioni** (analoga alla riduzione di un'espressione aritmetica/algebrica): **Haskell**, **ML**, **LisP**, …
- **Logici**: specificano una **formula logica** di **cosa** il programma deve calcolare. Il controllo è implicito: **Prolog**, …

In questo corso…

Graffieremo la superficie del paradigma **funzionale** e a **oggetti**:

- Vedremo il **C++** che estende il linguaggio C con gli oggetti;
- Vedremo le basi di **Haskell** come esempio di linguaggio funzionale;
- Confronteremo la soluzione di qualche problema in diversi paradigmi.

Lezione 1b:

Informazioni Pratiche

Approposito di questo corso…

Lezioni:
Martedì, 9:00-11:00 - aula **E** – modalità **BLENDED**.

Pagina web:
`https://twiki.di.uniroma1.it/twiki/view/LDPMat/WebHome`

Trovate:

- diario delle lezioni
- materiali vari
- codice
- vecchi progetti etc.
- queste slides

Esame

Colloquio orale:

discussione homework settimanali
domande su Haskell
domande su C++

Il colloquio viene fatto **su appuntamento**.

Lezione 1c:

In principio era il λ-calcolo

In principio era il λ-calcolo

Sintassi:

$M ::=$	x	variabile
	$\lambda x.\, M$	astrazione
	$(M\ N)$	applicazione

Astrazione e applicazione sono analoghe alla dipendenza di una funzione da un argomento e all'applicazione di una funzione a un argomento. **Occorre fare attenzione ai nomi delle variabili**!

Computazione:

> Un termine nella forma **$(\lambda x.\, M)\ N$** si dice **β-redex** e si β-riduce al termine M' che è **M in cui tutte le occorrenze di x sono state sostituite con N**.

Un termine senza β-redessi è detto in **forma normale** ed è un **valore**.

Esempi di λ-termini e computazioni

[Alonso Church, 1931]

Identità: I $\equiv \lambda x.x$ $\qquad\qquad \forall M$ ho che **I** $M \equiv (\lambda x.x)M \rightarrow M$

Cancellatori:

K (o **T** per TRUE)$\equiv \lambda xy.x$
O (o **F** per FALSE)$\equiv \lambda xy.y$
if x ***then*** M ***else*** $N \equiv \lambda x.x\ M\ N$

$$(\lambda x.x\ M\ N)\ \mathbf{T} \rightarrow \mathbf{T}\ M\ N \equiv (\lambda xy.x)M\ N \rightarrow M$$
$$(\lambda x.x\ M\ N)\ \mathbf{F} \rightarrow \mathbf{F}\ M\ N \equiv (\lambda xy.y)M\ N \rightarrow N$$

Compositori:

S $\equiv \lambda xyz.xz(yz)$ $\qquad\qquad$ **S K K** $\equiv$ **I**, infatti:

$$\mathbf{S\ K\ K}\ M \equiv\ (\lambda xyz.xz(yz))\ \mathbf{K\ K}\ M \rightarrow \mathbf{K}\ M\ (\mathbf{K}\ M) \rightarrow M$$

Duplicatori:

$$\boldsymbol{\omega} \equiv \lambda x.xx\ \text{ - }\ \boldsymbol{\omega}_3 \equiv \lambda x.xxx\ \text{ - }\ \boldsymbol{\Omega} \equiv \boldsymbol{\omega}\ \boldsymbol{\omega}\ \text{ - }\ \boldsymbol{\Omega}_3 \equiv \boldsymbol{\omega}_3\ \boldsymbol{\omega}_3$$

$$\boldsymbol{\Omega} \equiv (\lambda x.xx)\ \boldsymbol{\omega} \rightarrow \boldsymbol{\omega}\ \boldsymbol{\omega} \equiv \boldsymbol{\Omega}$$
$$\boldsymbol{\Omega}_3 \equiv (\lambda x.xxx)\ \boldsymbol{\omega}_3 \rightarrow \equiv \boldsymbol{\omega}_3\ \boldsymbol{\omega}_3\ \boldsymbol{\omega}_3 \equiv \boldsymbol{\Omega}_3\ \boldsymbol{\omega}_3$$

Numeri, Iteratori e Ricorsori

[Church/Kleene, 1936]

In λ-calcolo è facile scrivere ricorsori o iteratori, per esempio i **numerali di Church**:

$$\underline{n} \equiv \lambda sz.s(s(\ldots(sz)\ldots)) \qquad [n \text{ applicazioni}]$$

Esempio $\underline{0} \equiv \lambda sz.z \equiv \mathbf{O} \equiv \mathbf{F}$ $\qquad \underline{3} \equiv \lambda sz.s(s(sz)))$

succ $\equiv \lambda xsz.s(xsz)$ da cui:

succ $\underline{3} \equiv (\lambda xsz.s(xsz))\ \underline{3} \to \lambda sz.s(\underline{3}sz) \equiv \lambda sz.s(\lambda xy.x(x(xy))\ s\ z)$

$\to \lambda sz.s(s(s(sz))) \equiv \underline{4}$

Attenzione! Le s e z dentro $\underline{3}$ sono **variabili diverse** dalle s e z esterne di succ). Le variabili legate da un λ si **devono rinominare** per evitare confusioni di nomi.

Che funzione è: $\lambda xy.x$ succ y? A cosa riduce $\underline{n}$ succ $\underline{m}$?

Morale: I **numerali di Church** rappresentano i **numeri naturali**, ma sono al tempo stesso iteratori (cioè permettono di definire funzioni per **iterazione** e **ricorsione primitiva**)

Il Punto fisso del λ-calcolo

[Church/Kleene, 1936]

Teorema: Nel λ-calcolo esiste un termine **Y** tale che per ogni altro termine M, $\mathbf{Y}\ M \rightarrow M\ (\mathbf{Y}\ M)$.

Dim: Consideriamo $\theta \equiv \lambda xy.\underline{y}(xxy)$ e definiamo $\mathbf{Y}{=}\theta\theta$.
Abbiamo che:

$$\mathbf{Y}\ M \equiv (\theta\theta)M \equiv ((\lambda xy.\underline{y}(xxy)\ \theta)\ M \rightarrow (\lambda y.\underline{y}(\theta\theta y))\ M$$
$$\rightarrow M\ (\theta\theta M) \equiv M\ (\mathbf{Y}\ M) \qquad \square$$

Abbiamo chiamato **Y** il **combinatore di punto fisso di Turing**, nome che viene usualmente dato al **combinatore paradossale di Curry**:

$$\lambda f.\ (\lambda\, x.\underline{f}(x\ x))\ (\lambda\, x.\underline{f}(x\ x))$$

Lezione 1

That's all Folks…

…Domande?