Programmazione Funzionale

Luca Abeni

May 11, 2016

Programmazione Funzionale

- Paradigma imperativo: computazione come modifica di stato
  - Variabili modificabili: ambiente associa nome a variabile, memoria associa variabile a valore
  - Costrutto fondamentale: assegnamento
    - Modifica associazione fra variabile e valore
    - R-Value “contenuto” nella variabile
  - Direttamente mappabile su macchina di Von Neumann
- Paradigma funzionale $\rightarrow$ no stato / variabili modificabili
  - Solo espressioni e funzioni (no comandi)
  - No assegnamento
  - Computazione come riduzione / riscrittura di espressioni
    - Invece che modifica dello stato...

Computazione Funzionale

- No stato modificabile $\rightarrow$ no iterazione (loop)!
  - Basata su ripetere operazioni fino a che un predicato è vero
  - Predicato: funzione booleana dello stato... Stato non modificabile $\Rightarrow$ predicato sempre vero o sempre falso $\Rightarrow$ iterazione non ha senso!
- Invece che iterazione, Ricorsione!
- Esempi di linguaggi funzionali: ML (varie incarnazioni), Lisp (varie incarnazioni - Scheme), Haskell, ...
  - Fondamento teorico / matematico: λ-calcolo!

Funzioni: Definizione ed Applicazione

- Funzione: relazione fra dominio e codominio che associa ad un elemento del dominio un solo elemento del codominio
  - $f : \mathcal{X} \to \mathcal{Y}$
  - $f \subset \mathcal{X} \times \mathcal{Y} : (x, y_1) \in f \wedge (x, y_2) \in f \Rightarrow y_1 = y_2$
  - $(x, y) \in f \to y = f(x)$
- $f(x)$: significato ambiguo
  - $f(x) = x^2$: definizione di $f()$
  - $f(3)$: applicazione di $f()$ al numero 3
  - Stessa sintassi ($f(x)$) per esprimere la definizione e l'applicazione di una funzione?
- In matematica, il significato di "$f(x)$" dipende dal contesto...
- ...In un linguaggio di programmazione, vorremmo una semantica meglio definita!
- Distinguere definizione da applicazione: in ML, `fn` per definizione

Definizione di Funzioni

- In linguaggi di programmazione, sintassi per distinguere definizione da applicazione
  - Esempio: in ML, **fn** x => x * x; definisce una funzione che calcola x^2
- In linguaggi funzionali, le funzioni sono *valori esprimibili*
  - Esempio: **val** f = **fn** x => x * x; definisce un simbolo `f` associato alla funzione che calcola x^2
  - Nota: sul libro, definizione leggermente imprecisa di `fun` (è `val rec`, non `val`!)
- Conseguenza dell'esistenza di un costrutto per definire funzioni (`fn` in Standard ML) : si possono definire anche *funzioni anonime*
  - "**fn** x => x * x;", senza "**val** f =" davanti
- Curiosità: esiste qualcosa di simile anche in C++:

```
[](int x) {
  return x * x;
};
```

Applicazione di Funzioni

- Differenti linguaggi definiscono differenti sintassi
  - `f(2);`
  - `(f 2)`
  - `f 2;`
  - Nota: in ML tutte le espressioni di qui sopra sono valide...
- A differenza di matematica, chiara differenza fra definizione ed applicazione di una funzione!
- Interessante conseguenza: si possono applicare anche funzioni anonime
  - (**fn** x => x * x) 3;
- Ancora, anche in C++:

```
[](int x) {
  return x * x;
}(3)
```

Esecuzione Come Valutazione

- Programma funzionale: composto da espressioni
- “Eseguito” valutando le espressioni
- Esempio: fattoriale!

```
unsigned int fact(unsigned int n)
{
  return n == 0 ? 1: n * fact(n - 1);
}
```

  - Notare “if aritmetico” (`p ? a : b`)
  - `fact(4)` = ?

```
fact(4) = (4 == 0) ? 1 : 4 * fact(3) = 4 * fact(3) =
4 * ((3 == 0) ? 1 : 3 * fact(2)) = 4 * 3 * fact(2) =
4 * 3 * ((2 == 0) ? 1 : 2 * fact(1)) = 4 * 3 * 2 *
fact(1) =
4 * 3 * 2 * ((1 == 0) ? 1 : 1 * fact(0)) =
4 * 3 * 2 * 1 * 1 = 24
```

Alcune Osservazioni

- Per essere puramente funzionale, if aritmetico!

```
unsigned int fact(unsigned int n)
{
  return n == 0 ? 1: n * fact(n - 1);
}
```

vs

```
unsigned int fact(unsigned int n)
{
  if (n == 0) {
    return 1;
  } else {
    return n * fact(n - 1);
  }
}
```

- Esecuzione come valutazione/sostituzione possibile solo in assenza di effetti collaterali

Computazione Come Riduzione

- In linguaggi funzionali si parla spesso di riduzione / riscrittura invece che valutazione
  - Enfatizza il processo di *sostituzione testuale* di sotto-espressioni
  - In particolare, una funzione applicata ad un argomento è sostituita con corpo della funzione
    - Sostituendo il parametro formale col parametro attuale...
  - Ricorda qualcosa? (Hint: passaggio parametri per nome)
    - Ricordare problemi con cattura parametri, chiusure, etc...
- Esempio: se **val** f = **fn** x => x * x, allora f 2 sostituito con 2 * 2
- Si può vedere come manipolazione di stringhe (no variabili, no esecuzione, no stack...)

Esempio di Riduzione

- **val rec** fact = **fn** n => **if** n = 0 **then** 1 **else** n $*$ fact (n $-$ 1)

```
fact 4 →
(fn n => if n = 0 then 1 else n * fact (n - 1))4 →
if 4 = 0 then 1 else 4 * fact 3 →
4 * fact 3 →
4 * (if 3 = 0 then 1 else 3 * fact 2) →
4 * (3 * fact 2) →
4 * (3 * (if 2 = 0 then 1 else 2 * fact 1)) →
4 * (3 * (2 * fact 1)) →
4 * (3 * (2 * (if 1 = 0 then 1 else 1 * fact 0))) →
4 * (3 * (2 * (1 * (fact 0)))) →
4 * (3 * (2 * (1 * (if 0 = 0 then 1 else 0 * fact
-1)))) →
4 * (3 * (2 * (1 * 1))) → 24
```

Riduzione: Altro Esempio

```
val K = fn x => fn y => x;
val S = fn p => fn q => fn r => p r (q r);
val a = ...;
```

- A cosa valuta `S K K a`?

S K K a $\rightarrow$
(**fn** p => **fn** q => **fn** r => p r (q r)) K K a $\rightarrow$
(**fn** q => **fn** r => K r (q r)) K a $\rightarrow$
(**fn** r => K r (K r)) a $\rightarrow$
K a (K a) $\rightarrow$
(**fn** x => **fn** y => x) a (K a) $\rightarrow$
(**fn** y => a) (K a) $\rightarrow$
a

- Nota: possibilità di “riscritture all’infinito”
  - **val rec** r = **fn** x => r (r x)
  - Computazione “divergente”

Programmazione Funzionale: Concetti Fondamentali

- Come detto, no comandi
- Espressioni
  - Valori (irriducibili) ed operatori primitivi
- Concetti fondamentali: astrazione ed applicazione
  - Astrazione: da espressione `e` ed identificatore `x`, costruisce `fn x => e` (astrae `e` dallo specifico valore di `x`)
    - Funzione (anonima) che mappa `x` in `e`
  - Applicazione: da funzione `f` ed espressione `e`, costruisce `f e`. Applica `f` ad `e`, valutando il valore della funzione `f` in base al valore di `e`
    - Operazione inversa rispetto all'astrazione

Riduzione Rivista

- Riduzione di un'espressione: avviene usando 2 meccanismi principali
  1. Ricerca nell'ambiente (e sostituzione di identificatori con loro definizione)
  2. Applicazione di funzioni (usando passaggio di parametri per nome e regola di copia)
- Esempio di 1: S K K a $\rightarrow$ (**fn** p => **fn** q => **fn** r => p r (q r)) K K a
  - Sostituisce S con **fn** p => **fn** q => **fn** r => p r (q r) perché **val** S = **fn** p => **fn** q => **fn** r => p r (q r)
- Esempio di 2: (**fn** p => **fn** q => **fn** r => p r (q r)) K K a $\rightarrow$ (**fn** q => **fn** r => K r (q r)) K a
  - Sostituisce (**fn** p => **fn** q => **fn** r => p r (q r)) K con **fn** q => **fn** r => K r (q r) eliminando **fn** p => e sostituendo `p` con `K`

Linguaggi Funzionali: Caratteristiche Fondamentali

- Come detto, funzioni come valori esprimibili
- Omogeneità fra dati e funzioni
- Funzioni che ricevono funzioni come argomenti
- Funzioni che generano funzioni come risultato
  - Nota: sembra facile, ma ci sono complicazioni...
  - Ambiente della funzione ritornata?
  - Necessità di implementare chiusure!
- Spesso si parla di *funzioni di ordine superiore*...

Mettendo Tutto Assieme: Programmi Funzionali

- Programma: insieme di definizioni di valori
  - Modifica ambiente introducendo nuove associazioni
  - Possono richiedere valutazione di espressioni complesse
- Eseguito tramite riscrittura simbolica di stringhe (riduzione)
- Semplificazione successiva di un'espressione fino a forma semplice (valore) tramite 2 operazioni:
  - Ricerca di binding fra identificatori e valori nell'ambiente e sostituzione dell'identificatore con il valore trovato
  - Applicazione di funzioni ad argomenti (usando passaggio di parametri per nome)
    - Passaggio per nome: regola di copia $\rightarrow$ β-regola!

Applicazione di Funzioni: la β-Regola

- Si applica ad espressioni riducibili (*redex*)
  - Funzioni applicate ad argomenti
  - `(fn x => e) y`
- Ridotto di una redex `(fn x => e) y`:
  - In `e`, sostituisce ogni istanza di `x` con `y`
- β-regola: se `e` contiene una redex, riduci (semplifica) `e` ad `e1` sostituendo la redex col suo ridotto (`e` $\rightarrow$ `e1`)
- Fino a qui, definizioni informali:
  - Quando / come termina la riduzione? (cos'e' un'espressione irriducibile?)
  - β-regola: definizione formale? (cosa fare quando abbiamo più redex? ...)

Valori e Funzioni

- Valore: espressione non ulteriormente semplificabile
  - Valore di un tipo conosciuto dal linguaggio
  - Valore funzionale
- Funzioni come valori... Sorta di “computazioni ritardate”
  - Esempio: **val** G = **fn** x => ((**fn** y => y + 1) 2);
  - A cosa è legato il nome “`G`”?
  - Nella definizione della funzione `G` l’applicazione `(fn y => y + 1) 2` non viene valutata...
  - ...Oppure è valutata al valore `3`, per cui `G` è legato a `fn x => 3`?
- Espressioni **fn** x => e vengono valutate al momento dell’applicazione (β-regola), non della definizione!
- Ancora definizione “non troppo formale” di argomenti... Definizione formale verrà dopo!

Valutazione degli Argomenti

- Come applicare la β-regola in presenza di più redex
- Valutazione da sinistra a destra, ma...

```
val K = fn x => fn y => x;
val r = fn z => r(r(z));
val D = fn u => if u = 0 then 1 else u;
val succ = fn v => v + 1;
```

- Come si semplifica **val** v = K (D (succ 0)) (r 2);?
  - Quale redex si riduce per prima?
  - `K (D (succ 0)), D (succ 0)` o `succ 0`?
- Valutazione per valore, per nome o lazy

Valutazione per Valore

- Anche detta “in ordine applicativo”, “eager” o “innermost”
- Valuta prima redex “più interni”
- Un redex viene valutato se il suo argomento è un valore
- Algoritmo:

  1. Scandisci l’espressione da sinistra cercando il primo redex `f y`
  2. Semplifica (ricorsivamente) prima `f` fino a ridurla a **fn** x => e
  3. Poi semplifica (ricorsivamente) `y`, riducendola ad un valore `val`
  4. Applica la β-regola a (**fn** x => e)**val** e riparti dal punto 1

- Esempio precedente:
  - Redex più a sinistra `K (D (succ 0))`; `K`, `d` e `succ` non sono riducibili
  - Riduce quindi `succ 0 = (fn v => v + 1) 0 = 1`. Poi, `D 1 = (fn u => if u = 0 then 1 else u) 1 = 1`. Quindi, `K 1 = fn y => 1`, che è un valore.
  - Si è quindi ottenuto `(fn y => 1)(r 2)`; la funzione non è riducibile, riduci quindi l’argomento `(r 2)`. Valutando `(r 2)`, `r 2 = r(r 2) = r(r(r 2)) = ...` Computazione divergente!

Valutazione per Nome

- Detta anche “in ordine normale” o “outermost”
- Si valuta un redex prima del suo argomento
- Algoritmo:

  1. Scandisci l’espressione da sinistra cercando il primo redex `f y`
  2. Semplifica (ricorsivamente) prima `f` fino a ridurla a **fn** x => e
  3. Applica la β-regola a (**fn** x => e)y e riparti dal punti 1

- Esempio precedente:
  - Redex più a sinistra `K (D (succ 0))`; `K`, non è riducibile, quindi applica `(D (succ 0))` a `K`
  - Riduce `K (D (succ 0))` ad un valore funzionale: `(fn x => fn y => x)(D (succ 0)) = fn y => D (succ 0)`.
  - Quindi, `(fn y => D (succ 0))(r 2)`, che è un redex, ridotto a `D (succ 0)` (perché `y` non compare in `D (succ 0)` !!!). Questo equivale a `if (succ 0) = 0 then 1 else (succ 0)`.
  - Riducendo `succ 0 = 1` si ottiene `if 1 = 0 then 1 else (succ 0) = (succ 0)`. Risultato finale: `1`.

Valutazione Lazy

- Esempio precedente di valutazione per nome
  - Ad un certo punto si arriva a `if (succ 0) = 0 then 1 else (succ 0)`
  - `succ 0` viene valutata 2 volte!
  - Costo da pagare per non valutare l'argomento di una funzione
- Strategia di valutazione *lazy*: valuta ogni redex una sola volta
  - Quando si valuta un redex, si memorizza il valore
  - Se si incontra lo stesso redex da valutare ancora, si usa il valore memorizzato

Relazione fra Metodi di Valutazione

- Teorema:
  - Sia `e` un'espressione chiusa
    - Espressione chiusa: tutti i simboli sono legati tramite `fn`
  - `e` si riduce ad un valore primitivo `v` usando valutazione per valore o lazy, $\Rightarrow$ `e` si riduce a `v` usando valutazione per nome
    - Valore primitivo: valore non funzionale
  - valutazione per nome di `e` diverge $\Rightarrow$ la valutazione di `e` diverge anche usando la strategia per valore o lazy
- Notare che non è possibile valutare `e` ad un valore `v1` usando una strategia ed ad un valore `v2` $\neq$ `v1` usando una differente strategia
- Valutazione per nome "più potente" della valutazione per valore
  - Perché allora usare valutazione per valore?
  - Semplicità ed efficienza!