Tecniche di Programmazione Funzionale e Imperativa

Ivano Salvo

Inside Lazy Evaluation

Corso di Laurea in **Informatica**, III anno

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **10**, 8 aprile 2021

Lezione 10a:

Valutazione Lazy e valori

Call-by-name e valutazione lazy

Disclaimer: finora ho parlato di lazyness e call-by-name come due cose distinte, ma si tratta di fenomeni che originano insieme dalla strategia di riduzione leftmost-outermost.

Abbiamo già abbondantemente visto la strategia di riduzione **call-by-name** e abbiamo alluso "alluso" a due possibili impatti:

- l'**espressività** (possiamo considerare **funzioni non strette**) e
- l'**efficienza** (in molti casi la valutazione lazy **evita di fare conti inutili**).

Oggi entreremo nei dettagli di questi due aspetti.

Introduremo infine un ultimo beneficio: la possibilità di maneggiare in modo elegante, astratto ed efficace **strutture dati infinite**.

Nelle prossime lezioni ci divertiremo con la programmazione su liste infinite.

Valori in λ-calcolo

Nel λ-calcolo, oltre alle strategie di riduzione, è possibile scegliere diversi insiemi come **valori**, cioè **termini** che non è necessario **ridurre ulteriormente**.

Una scelta naturale sono le **forme normali**: sono i termini che non contengono redex. Ad esempio:

$$\mathbf{I}=\lambda x.x,\ \mathbf{K}=\lambda xy.x,\ \mathsf{due}=\lambda sz.s(s(x)),\ \mathbf{S}=\lambda xyz.xz(yz)$$

Un'altra scelta naturalmente collegata alle riduzioni di testa (esterne) è scegliere come valori le **forme normali di testa**. Un termine (chiuso) è una head-normal form se è nella forma:

$$\lambda x_1\ x_2\ \ldots.\ x_n.x_i\ N_1\ N_2\ \ldots N_m$$

Siccome i lambda termini sono **essenzialmente funzioni**, posso fare una scelta più estremista ed "estrarre" da un termine solo la sua dipendenza funzionale principale. Una **forma normale debole di testa** (o **weak head normal form**) ha la forma:

$$\lambda x.M$$

Non sono in **nessuna forma normale** $\mathbf{\Omega}=(\lambda x.xx)(\lambda x.xx)$ o $\mathbf{S\ K\ K} = (\lambda xyz.xz(yz))(\lambda xy.x)\ (\lambda xy.x)$ in quanto **applicazioni**, anche se **S K K** riduce a **I** e quindi **ha una forma** normale.

Valori in Haskell

Ovviamente Haskell eredita la stessa nozione di valore del λ-calcolo per quanto riguarda le funzioni. Essendo call-by-name, è naturale scegliere come valori le **weak-head normal form**. Quindi:

- una funzione è un valore (è un λ).
- Se applico una funzione definita a un argomento, sostituisco il nome con la sua defin. e riduco fino ad estrarre almeno un λ.

E per gli altri tipi di dato?

Una lista è in weak normal form se è nella forma `x:xs`, **indipendentemente** dal fatto che `x` e `xs` siano espressioni valutate! Sono perciò in weak head normal form:

`(3+2):merge [1,2,3][3,4]` e `undefined:undefined`

Per i tipi di dato definiti dall'utente si procede nello stesso modo, per cui sono weak-head normal form espressioni come:

```
(Just 3+2), (Just undefined), Succ (exp m n)
```

Lezione 10b:

Valutazione Lazy: Espressività

Funzioni non strette

Abbiamo già accennato al primo effetto immediato della valutazione lazy: abbiamo la possibilità di **definire funzioni non strette**, cioè funzioni che restituiscono un valore anche quando la valutazione di alcuni loro **parametri potrebbe non terminare**.

Fin dalla prima lezione abbiamo visto il cancellatore K.

Ovviamente questo fenomeno è interessante in quanto può verificarsi in forme estremamente "complicate".

```
K = \x y -> x
 -- Definisco una funzione non terminante
omega x = omega x
 -- Siccome K non dipende dal suo secondo argomento
K I (omega 4) 3
> 3
```

Funzioni non strette

Vediamo un esempio sempre minimale, ma forse più vicino alla pratica della programmazione. Consideriamo la funzione `null` che verifica se una lista sia vuota o meno.

Questa funzione si riduce prima che la lista sia completamente calcolata, non appena la lista raggiunge una weak head normal form. La prova, nei seguenti esperimenti.

Attenzione però, **non si riduce sempre**!

```
-- Questa funzione verifica se una lista è vuota
myNull [] = True
myNull  _ = False
 -- posso applicarla con successo a un oggetto
parziale, cioè non valutato o non valutabile
>null [undefined]
False
>null (undefined:undefined)
False
>null undefined
*** Exception: Prelude.undefined
```

strict or, call-by-value

I programmatori C sanno che il **seguente comando non genera mai** un errore di division-by-zero:

```
if (x==0 || mod(y,x)==2) …
```

perché se `x` valuta a `0`, la condizione valuta a `True` e non si va a valutare la seconda condizione (che con `x==0` significherebbe fare una divisione per 0).

Tuttavia in C **non è possibile** definire una funzione:

```
int myOr(int x, int y)
```

che si comporti come l'operatore `||` predefinito.

Il motivo è la **call-by-value del C** che **forza** sempre e comunque la **valutazione dei parametri**.

Ovviamente in Haskell è possibile e quindi è possibile avere lo stesso comportamento degli operatori predefiniti (continua…)

strict or e left or

Vediamo tre definizioni Haskell della funzione or (ce ne sarebbero anche altre di interessanti 😉).

Sono equivalenti? Come si comportano su `undefined`?

```
-- Questa funzione valuta il primo parametro.
-- se è True dà True, altrimenti valuta il secondo
leftOr True    _   = True
leftOr   _     x   = x

-- Questa funzione necessita di valutare sempre
-- entrambi i parametri
eagerOr True   True  = True
eagerOr False  True  = True
eagerOr True   False = True
eagerOr False  False = False

-- e questa?
whichOr False  False  = False
whichOr   _      _    = True
```

strict or, left or e right or

Si potrebbe dimostrare se e quando sono equivalenti: in un tipo finito come `Bool`, l'induzione corrisponde all'analisi per casi (sui casi base, in qualche senso) **tenendo conto di undefined**!.

Noi ci aiutiamo con l'interprete.

```
-- direi ovviamente…
> leftOr True undefined
True

-- altrettanto prevedibile dovrebbe essere:
> eagerOr True undefined
*** Exception: Prelude.undefined

-- forse un po' meno ovvio:
> whichOr True undefined
True
-- dipende dall'ordine di valutazione dei pattern

-- Infine, possiamo facilmente definire il rightOr
rightOr = (flip leftOr) --per pattern matching?
> rightOr undefined True
True
```

left or e parallel or

Il ruolo dei parametri però non è simmetrico, e infatti…

Si potrebbe infine desiderare una funzione che abbia il seguente comportamento (”*parallel or*” o **por** di PCF):

```
por undefined True = True
por True undefined = True
```

Questo in Haskell **non è possibile**, perché **l’interprete fissa un ordine di valutazione**. Per avere una tale funzione occorrerebbe parallelamente (da cui il nome) eseguire entrambe le espressioni, fermarsi non appena una delle due termina (e/o bloccando eventuali eccezioni e continuare con l’altro parametro) etc. etc.

```
-- purtroppo però
> leftOr undefined True
*** Exception: Prelude.undefined

-- e quindi ovviamente anche
> rightOr True undefined
*** Exception: Prelude.undefined
```

Lezione 10c:

Valutazione Lazy: Efficienza

Efficienza: Sharing

Le computazioni lazy **potrebbero essere meno efficienti**, nei casi in cui una funzione sia un **duplicatore**.

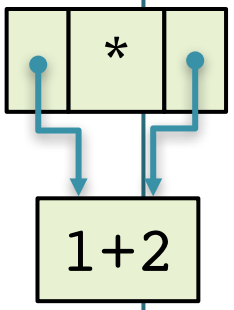
Questa cosa si risolve rappresentando i **termini con DAG invece che con alberi**, creando sharing quando so che due sotto-espressioni sono uguali.

Qualcuno definisce la lazy evaluation come la combinazione tra **call-by-name** e **sharing**.

Tuttavia, lasciare pointer a espressioni non valutate (**thunk**) ha un **costo in termini di memoria**!

```
 -- square produce due copie del suo parametro
square x = x * x

 -- vediamo una riduzione:
square (1+2) =
   = (1+2) * (1+2) = -- duplico la stessa expr
   = 3* (1+2) = -- qui ricalcolo la stessa expr.
   = 3*3
   9
```



Efficienza: valutazioni parziali

Ovviamente il rovescio della medaglia è che **valutare i parametri solo quando necessario** può viceversa avere grandi benefici computazionali.

Abbiamo "alluso" a questa possibilità nell'esempio del minimo intero libero, quando la funzione `head` **poteva estrarre** il primo intero libero **prima** di calcolare la lista di tutti gli interi liberi.

```
 -- differenza tra liste ordinate
 -- notare che si possono defin. operatori infissi
(x:xs)\\(y:ys)  |x==y = xs \\ ys
                | x<y = x: xs \\ (y:ys)
                | x>y = (x:xs)\\ ys
   []\\xs = []         xs\\[]=xs

head ([1,3,4,5,7]\\[1,2,3,5])   --clausola ==
   → head ([3,4,5,7]\\[2,3,5])  --clausola >
   → head ([3,4,5,7]\\[3,5]) --clausola ==
   → head ([4,5,7]\\[5])        --clausola <
   → head 4:([5,7]\\[5])        --head
   → 4   --la computazione di \\ non si completa
```

Laziness ed efficienza: foldl

Vediamo il problema di lasciare **espressioni non valutate**, con un grande classico: rivediamo la definizione di `foldl`.

Tutte le **espressioni temporanee** del calcolo sono tenute con pointer dentro i cosiddetti thunk. Vanno **deallocate** quando non servono più con **garbage collector**: **overhead in tempo e spazio**!

```
 -- Ricordiamo il caso di foldl
foldl f v   []   = v
foldl f v (x:xs) = foldl f (f v x) xs

foldl (+) 0 ([1,2,3]                -- notazione
  → foldl (+) 0 (1:2:3:[])          -- clausola rec-foldl
  → foldl (+) (0+1) (2:3:[])        -- clausola rec-foldl
  → foldl (+) ((0+1)+2)(3:[])       -- clausola rec-foldl
  → foldl (+) (((0+1)+2)+3)([])-- clausola []-foldl
  → ((0+1)+2)+3                     -- (+)
  → (1+2)+3                         -- (+)
  → 3+3                             -- (+)
  → 6                               normal form
```

Cosa desidereremo da foldl (+)

Almeno in questo caso, sarebbe meglio fare le operazioni non appena possibile.

Sarebbe più efficiente il calcolo seguente, in quanto i calcolatori eseguono efficientemente i calcoli aritmetici (meglio di allocazioni/deallocazioni di memoria)

Haskell ha una primitiva per forzare la valutazione:

```
seq:: a -> b -> b
```

con la semantica: per valutare `seq x y`, valuta `x` e poi valuta `y`.

```
foldl (+) 0 ([1,2,3]                    -- notazione
  → foldl (+) 0 (1:2:3:[])               -- clausola rec-foldl
  → foldl (+) (0+1) (2:3:[])             -- (+)
  → foldl (+) 1 (2:3:[])                 -- clausola rec-foldl
  → foldl (+) (1+2)(3:[])                -- (+)
  → foldl (+) 3 (3:[])                   -- clausola rec-foldl
  → foldl (+) (3+3)([])                  -- (+)
  → foldl (+) 6([])                      -- clausola []-foldl
  → 6                                 normal form
```

Eager Evaluation in Haskell

Nell'espressione `seq x y`, usualmente abbiamo che `y` dipende in qualche modo da `x`.

Vediamo una definizione alternativa di `foldl` che usa `seq`.

L'uso di seq usualmente è limitato al caso della **eager** (anche detta **strict**) **evaluation** che in Haskell si può fare con l'operatore infisso `($!)`.

Nota: In Haskell (f x) è equivalente a f $ x (senza !)

```
 -- Vediamo la foldl strict sui calcoli ricorsivi
foldl' f v   []   = v
foldl' f v (x:xs) =
        let z = f v x in z `seq` foldl' f z xs

 -- equivalentemente
foldl'' f v   []   = v
foldl'' f v (x:xs) = ((foldl' f) $! (f v x)) xs
```

Lezione 10c:

Valutazione Lazy: Strutture Dati infinite

Strutture Dati Infinite

Una interessante applicazione della valutazione lazy è la possibilità di maneggiare in modo **astratto** ed **elegante strutture dati infinite**, non altrettanto naturale coi linguaggi **eager**.

Noi vedremo un po' di programmi che generano **liste infinite**, chiamate anche **stream**.

```
 -- Possiamo definire la lista con infiniti 1
ones = 1:ones
 -- Ovviamente, se chiediamo di valutare ones
 -- ci vengono stampati infiniti 1...
> ones
[1,1,1,1,1,1,1,1,1,1,…
^C
 -- tuttavia…
> take 3 ones
[1,1,1]
 -- … termina correttamente. take è predefinita
myTake  _    []    = []
myTake  0    xs    = []
myTake  n (x:xs) = x : myTake (n-1) xs
```

Laziness: Espressioni non terminanti

Cerchiamo di vedere come opera la valutazione lazy e riduciamo `take 3 ones`.

```
take 3 ones
      -- per ridurre take devo trasformare ones
      -- nella forma x:xs
    → take 3 (1:ones)
      -- riduco sempre il redex più esterno,
      -- applicando la clausola per take
    → 1:take 2 ones
      -- come prima: sono costretto a ridurre ones
    → 1:take 2 (1:ones)
      -- e così via...
    → 1:1:take 1 ones
    → 1:1:take 1 (1:ones)
    → 1:1:1:take 0 ones → 1:1:1:[]
 --  -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
      -- Teniamo a mente la def di take:
myTake  _   []    = []
myTake  0   xs    = []
myTake  n (x:xs) = x : myTake (n-1) xs
```

Alcuni simpatici esempi…

Come definire lo stream dei numeri naturali?

Sappiamo che in Haskell possiamo indicarlo con lo zucchero sintattico: `[0..]`

Possiamo usare un generatore…

Oppure aiutarci con i nostri funzionali sulle liste…

Analogamente possiamo fare le potenze di un numero, ad esempio…

```
 -- Uso di un generatore (parametrico) per
 -- definire i naturali:
nats = nextNat 0 where
   nextNat n = n:nextNat (n+1)

 -- Forse più affascinante il seguente:
nats' = 0:map (+1) nats'

 -- Capito il trucco…
powers n = 1:map (n*) (powers n)
```

Lezione 10

That's all Folks…

Grazie per l'attenzione…

…Domande?

```
W(let x=\y->y in xx, {})->
   W(\y->y, {}) = a -> a
       W(y, {y:a}) = a, {}
   colpo di generalizzazione per tipare x

   x:\/a. a->a

   devo tipare xx nell'ambiente {x:\/a. a->a}
     il tipo della prima x deve essere t1->t2
     il tipo della seconda x deve essere t3
unificabile con t1

     x: b->b   x:c->c --> sostituzione {(b,c->c)}
     tra b e d (d fresca) --> {(b,d)}
       {(d,c->c)}
     U (b->b, (c->c)->d)

       La prima x: (c->c)->(c->c)

\/c.c->c
```