Tecniche di Programmazione Funzionale e Imperativa

Ivano Salvo

Proofs I

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

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **8**, 25 marzo 2021

Lezione 8a

Program Calculation

Confluenza e Call-by-Name

Abbiamo visto che le computazioni di Haskell sono **Church-Rosser**, cioè **non dipendono dall'ordine di valutazione** delle sottoespressioni (**a meno di non-terminazione**).

Abbiamo però visto che la strategia **call-by-name assicura** (quando possibile) la **terminazione**.

Queste due proprietà (vedremo entrambe false per la classica programmazione imperativa call-by-value, del C per esempio) sono l'ingrediente principale per poter sviluppare **dimostrazioni di equivalenza di programmi algebriche**.

Program Calculation I: undefined

Cominciamo con un curioso valore in Haskell.

Lo useremo, a volte, ad esempio quando ci servirà analisi di eventuali computazioni non-terminanti.

```
 -- valore undefined, unico abitante di ∀a.a
> :t undefined
undefined :: a
 -- ma ovviamente
> undefined
*** Exception: Prelude.undefined
 -- ma anche (riflettere sul tipaggio)
> :t undefined : undefined
undefined : undefined :: [a]
 -- so che si tratta di una lista...
```

Program Calculation II: append

Facciamo un piccolo esempio di calcolo (append o ++):

```
++::[a]->[a]->[a]
   []  ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)

 -- Calcoliamo il seguente:
[1,2]++[3,4,5]
   --  { notation }
(1:2:[])++(3:4:5:[])
   --  { prima clausola di ++ }
1:(2:[])++(3:4:5:[]))
   --  { prima clausola di ++ ancora}
1:(2:([]++(3:4:5:[])))
   --  { seconda clausola di ++ ancora}
1:(2:(3:4:5:[])))
   --  { notation }
[1,2,3,4,5]
```

Lezione 8b

Laws & Proofs

Algebra

Siamo abituati alle dimostrazioni algebriche in matematica.

Vediamo un semplice esempio che riguarda un noto prodotto notevole: $(a + b)(a - b)=a^2 - b^2$.

$(a + b)(a - b) =$ { distributività }

$a\,(a - b) + b(a - b) =$ { distributività }

$a^2 - ab + ba - b^2 =$ { commutatività di * }

$a^2 - ab + ab - b^2 =$ { annull. opposti }

$a^2 - b^2$

Osservate che l'applicazione di regole algebriche ha un impatto computazionale: il numero delle operazioni aritmetiche da eseguire può cambiare applicando trasformazioni algebriche.

Alcune funzioni sul tipo Nat

Vediamo una prima prova di una proprietà per programmi Haskell. Definiamo l'esponenziale sui Naturali. Usiamo i `Nat` perché evidenziano la struttura induttiva meglio degli `Int`.

Diamo un po' di definizioni, delle funzioni sui naturali.

Dimostriamo ora che:

`exp m (add p q) = mul (exp m p)(exp m q)`

```
add :: Nat -> Nat -> Nat
add   Zero    n = n
add (Succ m) n = Succ (add m n)

mul :: Nat -> Nat -> Nat
mul Zero     n = Zero
mul (Succ m) n = add n (mul m n)

exp :: Nat -> Nat -> Nat
exp x Zero       = Succ Zero
exp x (Succ n)   = mul x  (exp n)
```

Induzione sui Naturali

```
exp m (add p q) = mul (exp m p)(exp m q)
```

Conviene procedere per induzione (sul secondo parametro)

E quindi, la parte destra e sinistra sono uguali quando `q` è `Zero`, indipendente dai valori di `m` e `p`.

Vediamo il caso induttivo.

```
   -- proof: caso base, parte sx
exp m (add p Zero) -->   { def. di add}
exp m p

-- proof: caso base, parte dx
mul (exp m p)(exp m Zero)  -->   { def. di exp}
mul (exp m p)(Succ Zero)   -->   { def. di mul}
add (exp m p) (mul (exp m p) Zero) -->  { def. di mul}
add (exp m p) Zero         -->   { def. di add}
(exp m p)
```

Laws&Proofs: induzione sui Naturali

```
exp m (add p q) = mul (exp m p)(exp m q)
```

Ancora per induzione (sul secondo parametro), semplifichiamo prima la parte destra e poi la sinistra, come prima.

E quindi, la parte destra e sinistra sono uguali per l'ipotesi induttiva.

```
   -- proof: caso induttivo, parte sx
exp m (add p (Succ n))  -->   { def. di add}
exp m (Succ (add p n))  -->   { def. di exp}
mul m (exp (m (add p n)))

-- proof: caso base, parte dx
mul (exp m p)(exp m (Succ n))   -->   { def. di exp}
mul (exp m p)(mul m (exp m n))  -->   { commut. di mul}
mul m (mul (exp m p)(exp m p))

-- per induzione ho proprio che:
mul (exp m p)(exp m p)=exp (m (add p n))
```

map, head e undefined

Abbiamo visto le "leggi" (o equazioni) che dovrebbero essere soddisfatte da `fmap` nei funtori e da `pure` e `<*>` nei funtori applicativi.

Più in generale, possiamo costruire una ricca teoria algebrica dei programmi funzionali, facendo facili prove **algebriche** e/o **induttive**. Cominciamo con un semplicissimo esempio.

Si dice che `map` è **strict**, cioè: `map udefined = undefined`.

```
f . head = head . map f
   -- proof: vediamo che per ogni lista (x:xs)
f . head (x:xs) -->  { def. di head}
f x
   -- vediamo la parte destra:
head . map f (x:xs)      -->  { def di (.)}
head ( map f (x:xs))     -->  { def di map}
head ( f x : map f xs)   -->  { def di head}
f x
   -- occorre provare per []:
head [] = undefined = f (head [])
   -- no se f non è stretta!
```

liste:equazioni, equazioni, equazioni...

Potete provare a dimostrare invece che queste equazioni valgono sempre.

Queste leggi **non dipendono dal contenuto della lista**, e questo deriva dal fatto che sono funzioni **polimorfe** sugli elementi della lista.

Leggi di questo tipo si chiamano **trasformazioni naturali** (gergo che deriva dall'Algebra e dalla Teoria delle Categorie)

```
   -- prima determinare i tipi
f . tail = tail . map f    -- map è stretta!
map f . concat = concat . map (map f)
map f . reverse = reverse . map f
concat . map concat = concat . concat
```

reverse - 1

Cominciamo con il far vedere un fatto base sulla funzione reverse:

```
reverse (xs ++ ys) = reverse ys ++ reverse xs
```

stavolta applicheremo le equazioni ricorsive in entrambi i versi.

```
reverse (xs ++ ys) = reverse ys ++ reverse xs
   -- proof: induzione sulle liste, caso base
reverse ([] ++ xs) -->   { def. di ++}
reverse xs         -->   { def. di ++}
[] ++ reverse xs   -->   { def. di ++ (contrario)}
reverse [] ++ reverse xs --> { def. di reverse (contrario)}

   -- proof: passo induttivo
reverse ((x:xs) ++ ys)  -->   { def. di ++}
reverse (x:(xs ++ ys))  -->   { def. di reverse}
reverse (xs ++ ys) ++ [x] -->   {induzione}
reverse ys ++ reverse xs ++ [x]    -->   { assoc di ++}
reverse ys ++ (reverse xs ++ [x]) -->{ def di reverse}
reverse ys ++ reverse (x:xs)    -->{ def di reverse}
```

reverse 2

Continuiamo con un altro fatto evidente

```
reverse . reverse = id
```

Lo facciamo ancora per induzione

```
reverse (reverse xs) = xs
   -- proof: induzione sulle liste, caso base
reverse (reverse []) -->   { def. di reverse}
reverse []           -->   { def. di reverse}
[]
   -- proof: passo induttivo
reverse (reverse (x:xs))          -->   { def. di reverse}
reverse (reverse xs ++ [x])       -->   { fatto preced.}
reverse [x] ++ reverse (reverse xs) --> { induzione}
reverse [x] ++ xs                 -->   { def di reverse}
[] ++ [x] ++ xs                   -->   { def di ++}
[x] ++ xs                         -->{ def di ++}
x:xs
```

Lezione 8c

Derivazione di Programmi da Specifiche

Derivare programmi da proprietà

Abbiamo visto la versione efficiente di reverse che accumula i risultati nei parametri. Ma è possibile "derivare" il codice partendo dall'algebra?

Supponiamo di conoscere la tecnica di accumulare i risultati sui parametri e quindi cercare una funzione binaria reversEff. Quali equazioni deve soddisfare?

Dovrà essere una funzione più generale che "combina" (o esegue contemporaneamente) `reverse` e `++`. Possiamo quindi aspettarci che valga la seguente equazione:

```
reversEff xs ys = reverse xs ++ ys
```

Usando questa specifica per reversEff, scriviamo il codice, che a questo punto, sarà corretto per costruzione

Derivare programmi da proprietà

```
derivazione clausole ricorsive di reversEff
   -- caso base, prima lista vuota
reversEff [] ys     -- { specifica di reversEff}
reverse [] ++ ys    -- { def. di reverse}
[] ++ ys            -- { def. di ++}
ys                  -- { equazione base di reversEff}

   -- passo induttivo, (x:xs)
reversEff (x:xs) ys          -- { specifica di reversEff}
reverse (x:xs) ++ ys     -- { def di reverse}
reverse xs ++ [x] ++ ys  -- { assoc. di ++}
reverse xs ++ ([x] ++ ys) -- { assoc. di ++}
reverse xs ++ (x:ys)     -- { specifica di reversEff}
reversEff xs (x:ys)

   -- da cui le equazioni ricorsive:
reversEff   []   ys = ys
reversEff (x:xs) ys = reversEff xs (x:ys)
```

Morale della Favola

Possiamo scrivere versioni inefficienti, ma facilmente ottenibili ad esempio come composizione di altre funzioni.

Poi usare la **prima versione come specifica** e con manipolazioni algebriche/induttive trovare versioni più efficienti.

Le proprietà che vogliamo soddisfatte dalla funzione guidano la scrittura del codice.

Quando si programma su strutture dati come liste, alberi… sarà utile munirsi di **un'arsenale di funzioni** (in larga parte già viste, come `map`, `foldl`, `foldr`, `filter`, etc.) e delle **loro proprietà**.

Lezione 8d

fold & friends

Una variante: foldr1

In alcune funzioni, è un po' imbarazzante fornire il valore iniziale su cui cominciare la ricorsione.

L'esempio classico è il calcolo del minimo: il valore giusto sarebbe $+\infty$, ma la cosa è molto inelegante. Con gli `Int`, interi a precisione limitata potremmo scegliere `maxInt`, ma se la lista fosse di interi a precisione illimitata la cosa sarebbe vieppiù imbarazzante.

Risulta comoda quindi la seguente variante di `foldr`.

Osservate che il tipo di `f` è diverso da `foldr`.

Da cui, ovviamente, otteniamo `minimum` e `maximum` sulle liste.

```
 -- ovviamente non è definita su []
foldr1 f   [x]  = x
foldr1 f (x:xs) = f x (foldr1 f xs)
foldr1 :: (a -> a -> a) -> [a] -> a

minimum = foldr1 min
maximum = foldr1 max
```

foldr: fusion law

La **fusion law** ha la forma:

```
*     f . foldr g a = foldr h b
```

per opportune funzioni/valori `h` e `b`. Vediamo due istanze:

```
double . sum = foldr ((+).double) 0
length . concat = foldr ((+).length) 0
```

ricordate che `sum` (somma di una lista) e `concat` (scioglimento di una lista di liste in una lista) sono definite usando `foldr`.

L'idea è di **derivare le proprietà di `h` e `b`** facendo la prova induttiva sui casi `undefined`, `[]` e `(x:xs)` e derivare i vincoli necessari per far valere l'equazione *.

La fusion law è una sorta di forma di **induzione generalizzata preconfezionata** da applicare alle definizioni date per `foldr`.

fusion law 1 – undefined e [ ]

```
f . foldr g a = foldr h b
```

```
f (foldr g a undefined) =
```

```
f undefined { foldr è stretta perché decompone la lista }
```

```
foldr h b undefined =
```

```
undefined
```

Da cui deriviamo che per liste undefined, l'equazione vale per le funzioni strette, cioè tali che **`f undefined = undefined`**

```
f (foldr g a []) = f a
```

```
foldr h b [] = b
```

Da cui deriviamo che **`f a = b`** .

fusion law 2 – caso induttivo

```
f . foldr g a = foldr h b
```

```
f (foldr g a (x:xs))                    { def. di foldr }
      = f (g x (foldr g a xs))
foldr h b (x:xs)                        { def. di foldr }
      = h x (foldr h b xs)              { induzione }
      = h x (f (foldr g a xs))
```

Da cui deriviamo deve essere (per ogni x, y) :

```
f (g x y) = h x (f y)
```

Teorema (FUSION LAW PER FOLDR) Se:

1. `f` è stretta,
2. `f a = b`,
3. `f (g x y) = h x (f y)`

allora `f . foldr g a = foldr h b`.

fusion law 3 – applicazione

```
foldr f a . map g = foldr h a
```

Cerchiamo le condizioni su `f, g, a,` ricordando che:

```
map g = foldr ((:).g) []
```

1. `foldr f a` è stretta (in quanto `foldr` è stretta)
2. `foldr f a (map g []) = foldr f a [] = a = foldr h a []`
3.
```
foldr f a (map g (x:xs))                    { def. di map }
   = foldr f a (g x : map g xs)             { def. di foldr }
   = f (g x) (foldr f a (map g xs))

foldr h a (x:xs)                            { def. di foldr }
   = h x (foldr f a xs)
```

che quindi vale per `f (g x) y = h x y`, cioè **`f . g = h`**

Quindi abbiamo scoperto che:

```
foldr f a . map g = foldr (f . g) a
```

che può avere interessante utilità computazionale.

foldl e foldr - 1

Data una funzione binaria `(#)::a->a->a` e una costante `c::a`, abbiamo che:

```
foldr (#) e [x1,x2,x3] = x # (y # (z # e))
foldl (#) e [x1,x2,x3] = ((e # x) # y) # z)
```

Ricordiamo che `reverse = foldl (flip (:)) []`.

Da ciò dovrebbe essere chiaro che valgono le seguenti equazioni:

```
foldr f e xs = foldl (flip f) e (reverse xs)
foldl f e xs = foldr (flip f) e (reverse xs)
```

Procedendo come in precedenza, si può far vedere che, date due operatori binari `(#)` e `(@)` abbiamo che:

```
foldl (@) e = foldr (#) e
```

a patto che:

```
(x # y) @ z = x # (y @ z)
e @ x = x # e
```

foldl e foldr 1

Da cui, nel caso particolare in cui (#) sia un operatore **associativo** con **elemento neutro e**, si deriva:

```
foldl (#) e = foldr (#) e
```

anche se **non sempre le due funzioni hanno le stesse computazioni**, ad es., in termini di efficienza spazio/tempo.

Esempio:

```
foldl (++) [] [x, y, z] = (([] ++ x) ++ y) ++ z
```

costa $3m+2n+p$, mentre:

```
foldr (++) [] [x, y, z] = (x ++ (y ++ (z ++ [])))
```

costa $m+n+p$.

In genere **però `foldr` ha richieste di memoria superiori** perché esegue i conti al "ritorno" dalla chiamate ricorsive, lasciando non valutate espressioni più complesse.

scanl & scanr

Ci sono due funzioni che applicano una funzione a tutti i prefissi (`scanl`) e i suffissi (`scanr`) di una lista.

```
scanl (#) e [x,y,z] = [e,e#x,(e#x)#y,((e#x)#y)#z]

scanr (#) e [x,y,z] = [x#(y#(z#e)), y#(z#e), (z#e), e]
```

```
>:t scanl
scanl :: (b -> a -> b) -> b -> [a] -> [b]
> :t scanr
scanr :: (a -> b -> b) -> b -> [a] -> [b]

> scanl (flip (:)) [] [1,2,3]
[[],[1],[2,1],[3,2,1]]
> scanr (:) [] [1,2,3]
[[1,2,3],[2,3],[3],[]]
> scanr (+) 0 [1,2,3]
[6,5,3,0]
> scanl (+) 0 [1,2,3]
[0,1,3,6]
```

Derivare il codice di scanl – caso base

La funzione `scanl f` applica `foldl f` a tutti i segmenti iniziali di una lista (prefissi). È specificata dall'equazione:

```
scanl f e = map (foldl f e) . inits
```

Si tratta, nuovamente di una definizione che porta a una comportamento quadratico. Tuttavia, possiamo fare delle traformazioni. Partiamo al solito col caso base:

```
scanl f e [] = map (foldl f e (inits []) =
      = map (foldl f e [[]]) =
      = [foldl f e []] =
      = [e]
```

Derivare il codice di scanl – induzione

Passo induttivo:

```
scanl f e (x:xs) =
      = map (foldl f e (inits (x:xs))
      = map (foldl f e ([]:map (x:) (intits xs))
      = foldl f e [] : map (foldl f e . (x:))
                           (inits xs)
      = e : map (foldl f e . (x:)) inits xs (*)
      = e : map (foldl (f e x)) (inits xs)
      = e : scanl (f e x) xs
```

Da cui la definizione "lineare" di `scanl`:

```
scanl f e [] = [e]
scanl f e (x:xs) = e : scanl (f e x) xs
```

(*) Dimostrare che: `foldl f e . (x:) = foldl (f e x)`

Lezione 8

That's all Folks…

Grazie per l'attenzione…

…Domande?