Tecniche di Programmazione Funzionale e Imperativa

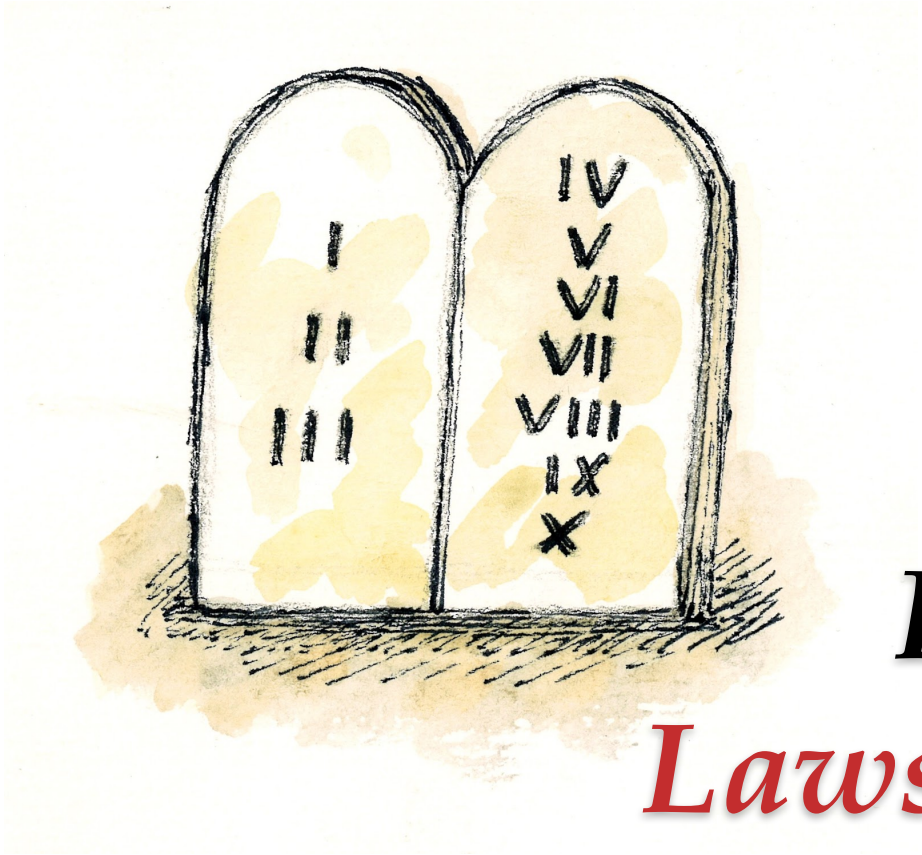
Ivano Salvo

L'onere e l'onore della Prova

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

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione 7, 15 marzo 2022



Lezione 7a

Laws & Proofs

Algebra

Siamo abituati alle dimostrazioni algebriche in matematica. Vediamo un semplice esempio di un noto prodotto notevole.

Osservate che le **trasformazioni algebriche** hanno un **impatto computazionale**: il **numero** delle **operazioni** aritmetiche da eseguire **può cambiare** applicando trasformazioni algebriche.

Ma ciò è stato determinato **simbolicamente** (una volta per tutte).

$(a + b)\,(a - b)$

$\quad = \{$ distributività $\}$

$a\,(a - b) + b\,(a - b)$

$\quad = \{$ distributività $\}$

$a^2 - ab + ba - b^2$

$\quad = \{$ commutatività di $\cdot\ \}$

$a^2 - ab + ab - b^2$

$\quad = \{$ annullamento degli opposti $\}$

$a^2 - b^2$

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

… $m^{p+q} = m^p \cdot m^q$ cioè che l'esponenziale trasforma somme in prodotti, ma lo facciamo **manipolando programmi Haskell**!

```
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 x n)
```

Induzione sui Naturali: Base

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

Conviene procedere per induzione su **q**

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

Caso Base: p=Z, parte sinistra

```
exp m (add p Z)
    = { def di add, caso Z}
exp m p
```

Ho applicato semplicemente le definizioni Haskell, ma a livello **simbolico**!

Caso Base: p=Z, parte destra

```
mul (exp m p) (exp m Z)
    = { def di exp, caso Z }
mul (exp m p) (S Z)
    = { def di mul, caso S }
add (exp m p) (mul (exp m p) Z)
    = { def di mul, caso Z }
add (exp m p) Zero
    = { def di add, caso Z }
exp m p
```

Passo Induttivo

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

Conviene procedere per induzione su **`q`**

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

Caso Induttivo: `p=S n`, sin.

```
exp m (add p (S n))
    = { def di add, caso (S n)}
exp m (S (add p n))
    = { def di exp, caso (S n)}
mul m (exp (m (add p n)))
```

Caso Induttivo: `p=(S n)`, destra

```
mul (exp m p) (exp m (S n))
    = { def di exp, caso (S n) }
mul (exp m p) (mul m (exp m n))
    = { comm. di mul }
mul m (mul (exp m p) (exp m n))
```

Ma adesso, le due parti verdi a destra e sinistra (in verde) sono **la proprietà da provare** per **`n`**, che è **strutturalmente più piccolo** di **`S n`**: quindi posso assumerle uguali applicando l'ipotesi induttiva.

CVD 😉.

Risvolti computazionali

Ma a cosa può essere utile questa proprietà? Scriviamola (per leggibilità) nelle usuali notazioni matematiche:

$$m^{p+q} = m^p \cdot m^q$$

Ovviamente quanto dimostrato per `Nat` vale per `Integer`…

Ma allora significa che la seguente funzione Haskell, **è corretta** per l'esponenziale (purtroppo abbiamo bisogno di due casi base, perché l'1 non si divide e vogliamo cmq farla funzionare per 0) rispetto al **programma ingenuo** che funge da **specifica** (abbiamo applicato la proprietà con p=q quando n=p+q **è pari**)

Abbiamo riscoperto la **Moltiplicazione Egiziana**!

```
exp :: Integer -> Integer -> Integer
exp m 0 = 1
exp m 1 = m
exp m n = mul (exp m p) (exp m q) where
   p = n div 2
   q = n - p
```

Induzione su Liste Finite

Associatività di **++**:
induzione su xs: osservate che le prove sono fatte usando il codice dei programmi!

Caso [ ]: (Caso Base)

```
([] ++ ys) ++ zs)
```
= { clausola (**1**) su primo ++}
```
ys ++ zs =
```
= { clausola (**1**) su "primo" ++}
```
[] ++ (ys ++ zs)
```

Caso x:xs: (Passo Induttivo)

```
((x:xs) ++ ys) ++ zs)
```
= { clausola (**2**) su primo ++}
```
(x:(xs ++ ys)) ++ zs =
```
= { clausola (**2**) su secondo ++}
```
x:((xs ++ ys) ++ zs) =
```
= { **Induzione** }
```
x:(xs ++ (ys ++ zs)) =
```
= { clausola (**2**) su primo ++}
```
(x:xs) ++ (ys ++ zs)
```

```
  []   ++ ys = ys              -- (1)
(x:xs) ++ ys = x:(xs ++ xs)    -- (2)
```

reverse è un'involuzione

Dimostriamo questa proprietà **per induzione su xs**.

```
reverse (reverse xs) = xs
```

Caso [ ]: (Caso Base)

```
reverse (reverse [])
   = { clausola (1) dentro ( ) }
reverse []
   = { clausola (1) }
[]
```

Caso x:xs: (Passo Induttivo)

```
reverse (reverse (x:xs))
   = { clausola (2) su interno}
reverse (reverse xs ++ [x]) =
   = { ??? }
```

… Non so cosa decomporre!

```
   = { ??? }
x:(reverse (reverse (xs)) =
   = { Induzione }
(x:xs)
```

Non si riesce a fare grandi progressi…
occorre un **Lemma** per induzione!

```
reverse   []   = []                  -- (1)
reverse (x:xs) = reverse xs ++ [x] -- (2)
```

Lemma su reverse

Dimostriamo una proprietà **più generale**.

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

Caso Base: `xs = []`:

```
reverse ([] ++ ys)
   = { def di ++ }
reverse ys
   = {def di ++ }
reverse ys ++ []
   = { reverse (←1) }
reverse ys ++ reverse []
```

Caso induttivo `xs = z:zs`:

```
reverse ((z:zs) ++ ys)
   = { def di ++ }
reverse (z:(zs ++ ys))
   = { clausola (2) su interno} reverse
reverse (zs ++ ys) ++ [z]
   = { induzione }
reverse ys ++ reverse zs ++ [z]
   = { associatività di ++}
reverse ys ++ (reverse zs++[z])
   = { clausola (←2) su interno} reverse
reverse ys ++ reverse (z:zs)
```

```
reverse   []   = []                  -- (1)
reverse (x:xs) = reverse xs ++ [x] -- (2)
```

map, head e undefined

Possiamo costruire una ricca teoria algebrica dei programmi funzionali, facendo facili prove **algebriche** e/o **induttive**.

Cominciamo con un semplicissimo esempio .

`f . head = head . map f` (se `f` **stretta**)

Si dice che `f` è **strict**, se `f udefined = undefined`: `map` è stretta, in quanto **decompone la lista**.

Caso Base: `xs = []`:

```
head [] =
   = { def di head }
undefined
   = { se f è stretta }
f (head [])
```

Caso induttivo (**sinistra**)

```
f . head (x:xs)
    = { def. di head}
f x
```

Caso induttivo (**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
```

osservare che si applica una logica lazy

map è un funtore

Proviamo la proprietà più importante di map:

```
map f . g = map f . map g
```

Caso Base (sinistra):

```
map (f . g)  [] =
   = { def di map }
[]
```

Caso Base (destra):

```
map f . map g  [] =
   = { def di . }
map f (map g [])
   = { def di map }
map f ([])
 = { def di map }
[]
```

Passo induttivo (sinistra/destra):

```
map (f . g) (x:xs)
    = { def di map}
(f . g) x : map (f . g) xs
    = { def di . }
f (g x) : map (f . g) xs
    = { induzione}
f (g x) : map f . map g xs
    = { def di map ←}
map f  (g x : map g xs)
    = { def di map ←}
map f (map g (x:xs)
    = { def di . ←}
map f . map g (x:xs)
```

liste: equazioni, equazioni, equazioni...

Potete provare a dimostrare che queste equazioni valgono sempre.

Queste leggi **non dipendono dal contenuto della lista**, e questo deriva dal fatto che sono funzioni **polimorfe** sul tipo dei valori contenuti nella 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 --f stretta!
   map f . concat = concat . map (map f)
     map f . reverse = reverse . map f
   concat . map concat = concat . concat
```

Induzione su Liste Parziali

Una **lista parziale** è `undefined` oppure `x:xs`, per qualche **x** e una **lista parziale xs** (una lista parziale non termina con `[]`).

Principio di induzione per **liste parziali**: Per dimostrare una proprietà *P* su liste parziali devo dimostrare:

- P(**undefined**)
- Se **xs** è parziale, faccio vedere che *P*(**xs**) implica *P*(**x:xs**)

Vediamo un esempio di prova per induzione su liste parziali.

`xs ++ ys = xs` (se `xs` **parziale**)

Caso Base:

```
undefined ++ ys =
   = { def di ++ }
undefined
```

Caso Induttivo (sinistra):

```
(x:xs) ++ ys =
   = { def di ++ }
x : (xs ++ ys)
   = { induzione}
x:xs
```

esempi su liste parziali: ++ e reverse

Alcune delle proprietà viste finore **non valgono** su liste parziali!

Vediamo se vale **`reverse (reverse xs) = xs`** nel caso in cui **`xs`** sia una lista parziale.

Cominciamo dal caso in cui **`xs = undefined`**, e scopriamo subito che **`reverse undefined = undefined`**, semplicemente perché reverse decompone la lista.

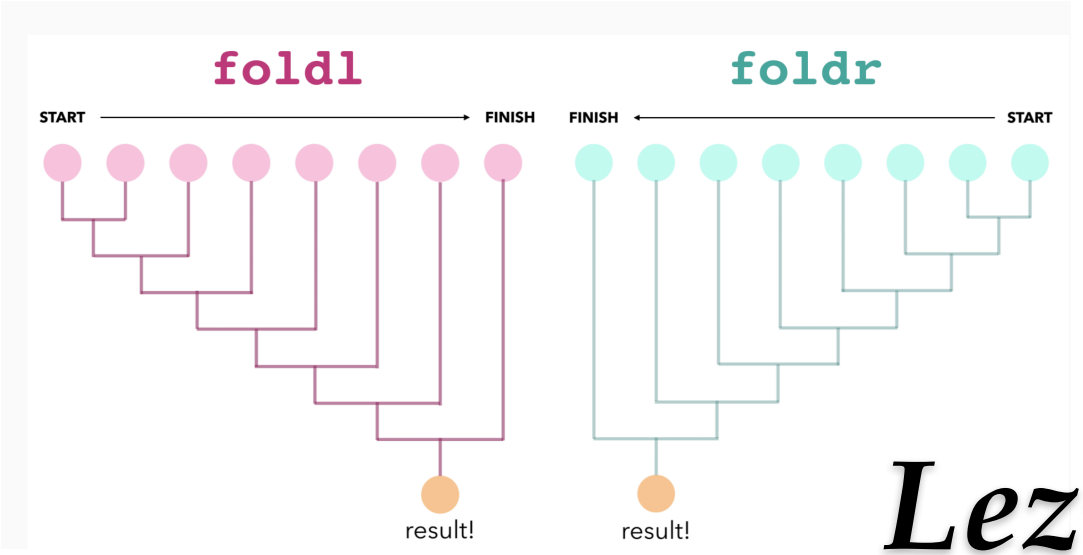
Quindi **`reverse . reverse`** è equivalente all'identità solo sulle liste finite!

Ma questo implica che **`reverse xs = undefined`** per ogni lista parziale…

perché **`undefined`** si propaga usando **`reverse`**. Infatti…

Caso Induttivo (sinistra):

```
reverse (x:xs) =
   = { def di reverse }
reverse xs ++ []
   = { induzione}
undefined ++ []
 = { ++ su liste parziali}
undefined
```



Lezione 7b

foldr & friends

Ricordiamo la funzione foldr

Come visto, **`foldr`** generalizza tutte quelle ricorsioni in cui si 'raccolgono' e calcolano i risultati al rientro dalla ricorsione.

Ricordiamo un po' di casi (ho aggiunto **`filter`**)

La figura può essere evocativa di cosa faccia foldr con una funzione binaria generica **`#`** e un valore **`v`**:

```
                [x, y, z] = x : (y : (z : []))
foldr (#) v [x, y, z] = x # (y # (z # v ))
```

cioè **`foldr`** sostituisce **`:`** con **`#`** e **`[]`** con **`v`**.

```
foldr ::  (a -> b -> b) -> b -> [a] -> b
myFoldr f v (x:xs) = f x (myFoldr f v xs)
myFoldr f v   []   = v
-- ad esempio:
mySum = foldr (+) 0
myLength = foldr (\x->(+1)) 0
myConcat = foldr (++) []
myFilter =
   foldr (\x xs -> if p x then x:xs else xs) []
```

Proprietà di foldr: "distributività"

Su tutte le definizioni che usano **`foldr`** date finora, vale una proprietà notevole:

```
foldr f e (xs ++ ys) = foldr f e xs # foldr f e ys
```

cioè, **`foldr`** "**distribuisce**" **`#`** su **`++`**. Cerchiamo di vedere se è sempre vero o se dipende dalle operazioni **`f`** ed **`e`**. Oppure se vale solo sotto precise condizioni e vediamo di determinarle.

Induzione! Vediamo il caso base:

Caso Base (sinistra):

```
foldr f e ([] ++ ys)
   = { def di ++ }
foldr f e ys
```

Caso Base (destra):

```
foldr f e [] # foldr f e ys
   = { def di foldr }
e # (foldr f e ys)
```

da cui deriviamo che **`e`** deve essere l'**elemento neutro** (**sinistro**) dell'operazione **`#`**.

Proprietà di foldr: "distributività"

```
foldr f e (xs ++ ys) = foldr f e xs # foldr f e ys
```

Vediamo il caso **induttivo**.

Caso induttivo (sinistra):

```
foldr f e ((x:xs) ++ ys)
   = { def di ++ }
foldr f e (x: (xs ++ys))
   = { def di foldr }
f x (foldr f e (xs ++ ys))
   = { induzione }
f x (foldr f e xs # foldr f e ys))
```

Caso Induttivo (destra):

```
foldr f e (x:xs) # foldr f e ys
   = { def di foldr }
f x (foldr f e xs) # foldr f e ys
```

da cui deriviamo che l'uguaglianza è soddisfatta se f e # soddisfano la proprietà:

```
f x (y # z) = (f x y) # z
```

che in particolare è vera se **f = #** e inoltre **# è associativo**!

Esempi

Siccome è noto che **+** (sui numeri) e **++** (sulle liste) sono **associativi** con **elementi neutri** 0 e **[]**, possiamo immediatamente derivare dalla proposizione precedente e dalla definizione di **sum** e **concat** con **foldr**, che valgono le uguaglianze:

```
sum (xs ++ ys) = sum xs + sum ys
concat (xss ++ yss) = concat xss ++ concat yss
```

Morale: **foldr** non è semplicemente un **meccanismo generalizzato di ricorsione**, ma offre un **meccanismo di induzione generale** sulle liste con cui **schemi di proprietà di programmi**.

Ultimo esempio: **filter**. Ricordiamo:

```
filter p = foldr (\x xs -> if p x then x:xs else xs)
```

e si può facilmente vedere che:

```
if p x then x:(ys++zs) else ys++zs
      =(if p x then x:ys else ys) ++ zs
```

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
```

ricordando che **sum** (somma di una lista) e **concat** (scioglimento di una lista di liste in una lista) sono definibili 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 (*) come appena fatto.

La fusion law è un'altra forma di **induzione generalizzata preconfezionata** da applicare alle definizioni date per **foldr**.

fusion law: undefined e [ ]

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

Caso [ ] (sinistra):

```
(f . foldr g a) []
   = { def di . }
f (foldr g a [])
   = { def di foldr }
f a
```

Caso [ ] (destra):

```
foldr h b []
   = { def di foldr }
b
```

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

Caso undefined (sinistra):

```
(f . foldr g a) undefined
   = { def di . }
f (foldr g a undefined)
   = { def di foldr }
f undefined
```

Caso undefined (destra):

```
foldr h b undefined
   = { def di foldr }
undefined
```

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

fusion law: caso induttivo

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

Caso (x:xs) (sinistra):

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

Caso [ ] (destra):

```
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 che **`f (g x y) = h x f y`** .

E quindi abbiamo il seguente:

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: applicazione

Vediamo se e quando vale la relazione:

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

Cerchiamo le condizioni su **`f`**, **`g`**, **`a`**. È un caso di fusion-law perché **`map g`** può essere scritta in termini di **`foldr`** e **`g`** [vedi *Homework*].

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)
```
uguali per induzione

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.

La sorellina di foldr: foldl

Come visto, **`foldr`** generalizza tutte quelle ricorsioni in cui si 'raccolgono' e calcolano i risultati al rientro dalla ricorsione.

Come visto, **`foldl`** generalizza tutte quelle ricorsioni in cui si **spingono in avanti** valori durante una discesa ricorsiva.

"Iterativamente", possiamo scrivere:

```
foldl (#) v [x, y, z] = ((v # x) # y) # z)
```

Facciamo i miei due pet-examples sul tema:

```
sommaPrec [2, 7, 4] = [0, 2, 9, 13]
sommaSucc [2, 7, 4] = [13, 11, 4, 0]
```

```
foldl ::  (b -> a -> b) -> b -> [a] -> b
myFoldl f v (x:xs) = myFoldl f (f v x) xs
myFoldl f v   []   = v
```

La sorellina di foldr: foldl

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

```
foldr (#) v [x, y, z] = x # (y # (z # v))
foldl (#) v [x, y, z] = ((v # x) # y) # z)
```

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
```

relazioni tra foldl e foldr

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

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

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

Esempio:

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

costa $3m+2n+p$, perché **++** è **lineare nel primo parametro**, mentre:

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

costa $m+n+p$.

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

Lezione 7

That's all Folks…

Grazie per l'attenzione…

…Domande?