Tecniche di Programmazione Funzionale e Imperativa

Ivano Salvo

L'Importanza di Essere Lazy
Efficienza ed espressività

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

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **8**, 18 marzo 2022

Lezione 8a:
Inside Lazy Evaluation

Call-by-name e valutazione lazy

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

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

Oggi entreremo nei dettagli di questi due aspetti.

Infine, un ultimo beneficio della laziness è la possibilità di maneggiare in modo elegante, astratto ed efficace **strutture dati infinite**.

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

Back to λ-calculus: valori

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

Una scelta naturale sono le **forme normali** (acronimo **nf**): 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 naturale, collegata alle riduzioni di testa (esterne) è scegliere come valori le **forme normali di testa**. Un termine (chiuso) è in **head-normal form** (acronimo **hnf**) se è nella forma:

$$\lambda x_1\ x_2\ \ldots.\ x_n.\boldsymbol{x_i}\ N_1\ N_2\ \ldots N_m\ (N_i \text{ non necess. in } nf)$$

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

$$\lambda x.M$$

Non **è forma normale** $\boldsymbol{\Omega}=(\lambda x.xx)(\lambda x.xx)$ o **S K K** (in quanto **applicazioni**) ma **S K K** riduce a **I** e quindi **ha una forma normale**. $\lambda y.\boldsymbol{\Omega}$ è in **whnf** (senza avere altre forme normali)

Valori in Haskell

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 a **estrarre almeno un λ**.

E per gli altri tipi di dato?

Una lista è in **weak head 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 anche `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)
```

Valori e valutazioni: esempi

Quanto venga poi valutato un dato, dipende dalla funzione che lo usa: ad esempio:

`print` richiede di ridurre a forma normale (= calcolare il valore) per stampare.

`head` chiede di ridurre una lista a weak head normal form ed estrae la testa (non necessariamente valutata)

`fst` sulle coppie fa la stessa cosa, per cui `fst (3+2, undefined)` riduce a `3+2` chiede di ridurre una lista a weak head normal form ed estrae la testa (non necessariamente valutata)

Funzioni come **`take n`** srotolano una lista finché trovano n elementi (ma non valutano le n espressioni!). Ad esempio:

```
length (take 2 [undefined, undefined, undefined])
```

valuta tranquillamente a 2! anche se stampare la lista dà errore. Ma è la stampa che forza la valutazione!

È bene ricordare che per fare **pattern-matching** è necessario valutare l'argomento, finché il pattern è riconoscibile (**non oltre!**)

Call by Name, whnf… e poi?

Abbiamo visto due ingredienti fondamentali della laziness:

- **call-by-name**: ricordiamo, valutare prima i **redex esterni** più a **sinistra** (leftmost-outermost).
- **fermarsi** alle **weak-head normal form** se non ci sono buoni motivi per andare oltre.

C'è un terzo ingrediente, che è lo **sharing di sottoespressioni**. Introduciamolo con un esempio.

```
sqr (sqr (3+4))
   = { def. di sqr (esterno) }
sqr (3+4) * sqr (3+4)
   = { def. di sqr (sinistra) }
(3+4) * (3+4) * sqr (3+4)
   = { def. di + (sinistra) }
7 * (3+4) * sqr (3+4)
   = 7 * 7 * sqr (3+4)
   = 49 * sqr (3+4)
   = … = { gli stessi conti parte sinistra }
2401
```

call-by-name

```
sqr (sqr (3+4))
   = { def. di + (interno) }
sqr (sqr (7))
   = { def. di sqr (interno) }
sqr (7*7)
   = { def. di * }
sqr (49)
 = { def. di sqr }
49*49 = 2401
```

call-by-value

```
sqr x = x * x -- duplicatore
```

sharing di sottotermini

Molte funzioni **duplicano i loro parametri**: ciò potrebbe causare un'enorme proliferazioni di **riduzioni ripetute** nella **call-by-name** che ritarda la valutazione dei parametri.

A questo, si somma il fatto di dover lasciare **molte espressioni in forma non-valutata** porta a **occupazione di memoria**.

Vediamo il **rimedio** per il **primo problema**.

```
sqr (sqr (3+4))
   = { def. di sqr (esterno) }
let y = sqr (3+4) in y*y
   = { def. di sqr (interno) }
let z = (3+4) in (let y = z*z in y*y)
   = { def. di sqr (interno) }
let z = 7 in (let y = z*z in y*y)
   = { def. di let esterno}
let y = 7*7 in y*y
   = { def di * }
let y = 49 in y*y
   = { def. di let }
49 * 49 = 2401
```

Stavolta **non ci sono conti ripetuti**, pur nel rispetto della **call-by name**.

Lazyness =
call-by-name
+ whnf
+ sharing

```
-- forma interna di Haskell equivale a:
sqr x = let y = x
        in y * y -- sharing argomento
```

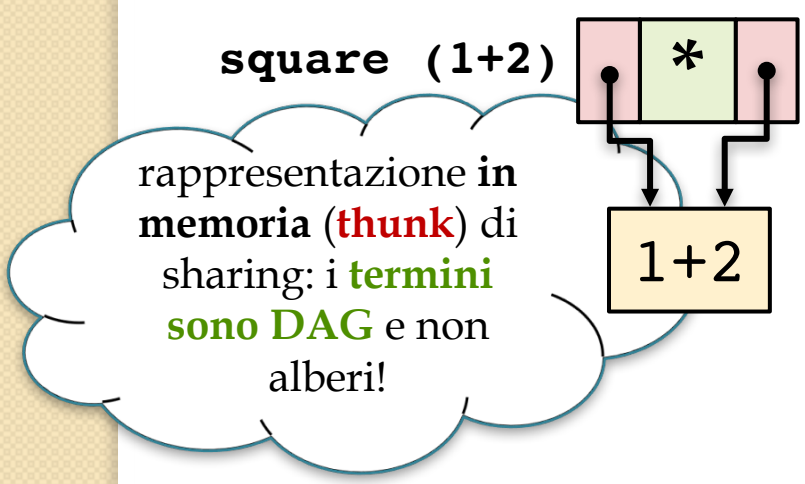
call-by-name inefficiente?

Facendo sharing, ho fatto **lo stesso numero di riduzioni** del caso call-by-value.

Regola aurea: una valutazione lazy **non fa mai più riduzioni** della stessa valutazione **eager** (o **strict**).

Vantaggio: può farne tante (anche **infinitamente**!) di meno.

Svantaggio: può lasciare molte **sottoespressioni non valutate** e ciò causa (a volte) un enorme **space leakage**!




```
-- 1000 riduzioni in meno…
> head (map (*) 2 [1..1000])
2
-- infinite meno riduzioni!
ones = 1 : ones
> ones
[1, 1, 1, 1, 1, ^C Interrupted
> head ones
1
```

Confluenza (Church-Rosser)

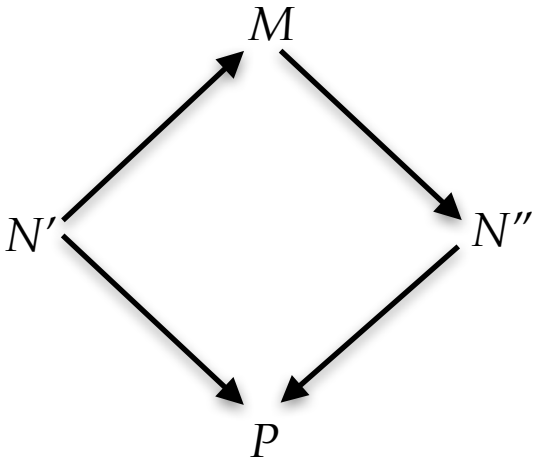
Tutte le riduzioni però hanno **dato lo stesso risultato indipendentemente dall'ordine**.

Non è un caso fortunato. In λ–calcolo (e Haskell) vale il seguente:

Teorema [CHURCH–ROSSER] *Se* $M \rightarrow^* N'$ *e* $M \rightarrow^* N''$ *allora esiste un termine* P *tale che* $N' \rightarrow^* P$ *e* $N'' \rightarrow^* P$

Il fatto che P esista **non significa che lo troviamo**! Tuttavia, se **tutte le riduzioni** di M **terminano**, necessariamente ***P* è un valore** (in forma normale) e **viene raggiunto da tutte le riduzioni**.

Corollario [UNICITÀ DELLE FORME NORMALI] *Se* N' *ed* N'' *sono forme normali e* $M \rightarrow^* N'$ *e* $M \rightarrow^* N''$ *allora* $N' = N''$.



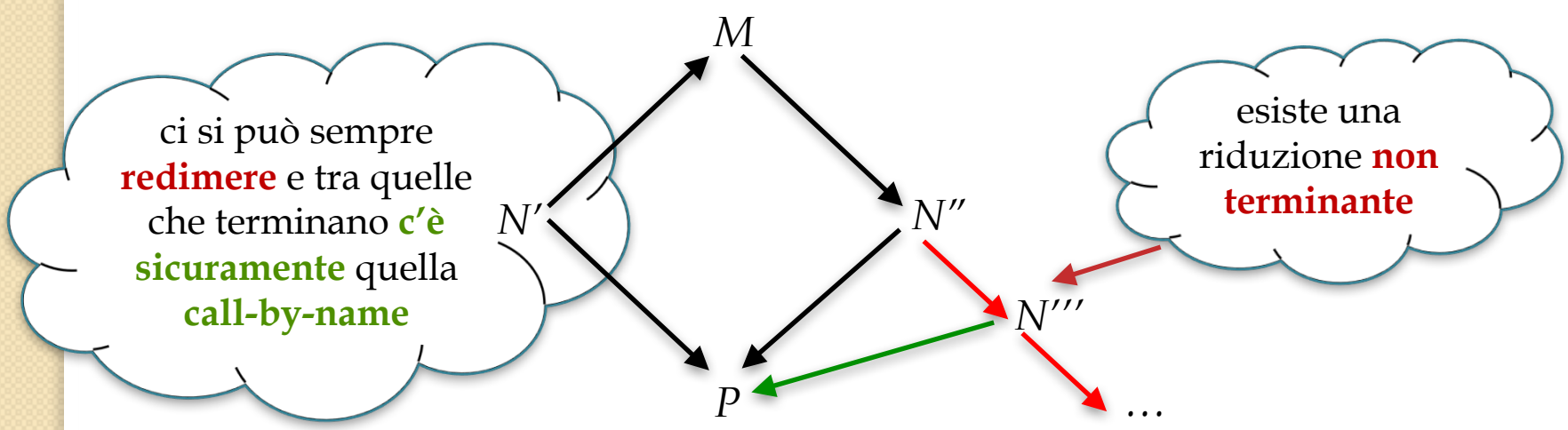
call-by-name è la λ-semantica naturale

Non è detto che troviamo le forme normali perché ci potrebbero essere computazioni non terminanti.

L'esempio canonico è il termine **K I Ω**: se io mi incapponisco a ridurre il termine **Ω**, non faccio mai progressi, mentre con un solo passo di riduzione avrei **K I Ω** → **I**.

Guarda caso, quella che termina, è la riduzione call-by-name. Ovviamente **non è un caso**. Vale il seguente:

Teorema [STANDARDIZZAZIONE] *Se esiste una riduzione $M \to P$ e P forma normale, allora la computazione call-by-name (leftmost-outermost) è tale per cui $M \to_{cbn} P$ in un numero finito di passi.*



Lezione 8b:
Espressività della valutazione Lazy

Funzioni non strette

Abbiamo già accennato al primo effetto immediato della valutazione lazy: possiamo 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** (ci sono solo 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**: esplora i **pattern** dall'**alto verso il basso** e i parametri da **destra verso sinistra**, a meno che non siano pattern che **non richiedono valutazione** (come una **variabile** o **_**)

Per avere una tale funzione occorrerebbe **parallelamente** eseguire entrambe le espressioni, **fermarsi** non appena **una delle due termina** (e bloccando eventuali **eccezioni**)

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

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

ancora sul parallelismo

Dovreste convincervi che valgono per `take` le seguenti equazioni:

```
take undefined [] = []          (L)
take 0 undefined = []           (R)
```

È possibile scrivere **`take`** in modo che soddisfi sia (**L**) che (**R**)?

No! Dobbiamo **decidere quale parametro valutare per primo**.

`lTake` e **`rTake`** giocano **sull'ordine** con cui **Haskell valuta** le **clausole** e **parametri** (che può essere esplicitato con l'**`if`**).

- **Esercizio**: Ma se avessi **`parallelOr`**?
- **Esercizio**: Definire **`pIf`** che soddisfa **`pIf undefined v v = v`** usando **`parallelOr`**. E viceversa.

```
-- questa soddisfa (L)
lTake 0  xs = []
lTake _  [] = []
lTake n (x:xs) =
      x : lTake (n-1) xs
```

```
-- e questa (R)
rTake _  [] = []
rTake 0  xs = []
rTake n (x:xs) =
   x : rTake (n-1) xs
```

Lezione 8c:
Tecniche di Ottimizzazione

Tecniche di "Ottimizzazione"

Abbiamo forse già visto che è possibile far comunicare diverse chiamate ricorsive usando **parametri** e **valori di ritorno**.

Questa lezione vedremo in dettaglio tecniche di ottimizzazione basate su **tupling** (ritornare più valori di quelli strettamente necessari) e **parametri accumulatori**.

Moralmente è come introdurre un **piccolo stato** di valori calcolati durante la computazione da condividere tra più chiamate ricorsive.

Vedremo in generale anche alcuni **limiti** della Programmazione Funzionale e in generale, la **maggiore difficoltà** di **prevedere** il **comportamento** dei **programmi**.

Questo può essere anche visto **in positivo**, nel senso che FP effettivamente offre al programmatore **potenti astrazioni**!

Un ricordo del prof. Corrado Böhm

"La programmazione funzionale è stateless, ma i parametri delle funzioni sono stateful"

si possono usare i parametri per **accumulare i risultati parziali** di una computazione

inaugurazione del primo calcolatore "italiano" allo IAC (~1955)

Parametri accumulatori

Cominciamo con un problema classico: la funzione **reverse**.

La versione immediata di reverse (vedi sotto) **è quadratica**, mentre il problema di **rovesciare** ad esempio una **pila di fogli** è intuitivamente **lineare**.

Il problema è l'asimmetria tra **:** e **++**: aggiungere in testa è costante, mentre **appendere in coda è lineare** nella lunghezza della prima lista.

Quindi **reverse** costa $1 + 2 + \ldots + (n\text{-}2) + (n\text{-}1) = \Theta(n^2)$.

Occorre fare **inserzioni in testa**, all'**"andata"**.

```
 -- reverse "naturale"
reverse    []    = []
reverse (x:xs) = reverse xs ++ [x]
```

$\Theta(n^2)$

$\Theta(1)$

$\Theta(n)$, n=length xs

Parametri accumulatori

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 **`++`**. Dobbiamo quindi aspettarci che valga la seguente equazione (**intuizione**: su `ys` ho già fatto il lavoro ed è **già la parte rovesciata**):

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

Usando questa specifica per **`reversEff`**, scriviamo il codice, **derivando** le **equazioni ricorsive** dalle **proprietà che deve soddisfare**.

Quindi il codice sarà **corretto per costruzione** e useremo la reverse ingenua come **specifica** (nell'equazione sopra).

Derivare programmi da proprietà

```
reversEff [] ys
   = { specifica di reversEff }
reverse [] ++ ys
   = { def di reverse }
[] ++ ys
   = { def di ++ }
ys
```

```
reversEff (x:xs) ys
   = { specifica di reversEff }
reverse (x:xs) ++ ys
   = { def di reverse }
reverse xs ++ [x] ++ ys
   = { associatività di ++ }
reverse xs ++ ([x] ++ ys)
   = { def. di ++ }
reverse xs ++ (x:ys)
   = { specifica di reversEff }
reversEff xs (x:ys)
```

Da cui, quelle scritte in rosso sono equazioni per **pattern matching** e si derivano le equazioni per **reversEff** di complessità **lineare**:

```
 -- reverse "efficiente"
reverse xs = reversEff xs [] where
   reversEff   []   ys = ys
   reversEff (x:xs) ys = reversEff xs (x:ys)
```

$\Theta(n)$

$\Theta(1)$

$\Theta(1)$

back to reverse

Dovrebbe essere chiaro che **reverse** è una sorta di **foldr** di **(++)** (circa perché chiaramente **occorre invertire i parametri** e rendere lista il primo parametro).

Usando il contrario di **cons** (cioè **(:)**) che chiameremo **snoc**, possiamo scriverla **più elengantemente**.

Mentre **reversEff** si può ottenere con il **foldl** di **(:)** (circa perché anche qui occorre invertire il ruolo dei parametri).

Più elegantemente lo possiamo fare con il funzionale **flip**.

```
reverse    = foldr (\x xs -> xs ++ [x]) []
-- il simpatico snoc
snoc x xs = xs ++ [x]
reverse'   = foldr snoc []
reverseEff = foldl (\xs x -> x:xs) []
-- ma ricordandosi di flip che inverte gli argomenti
-- abbiamo una formulazione più elegante:
reverseEff' = foldl (flip . (:)) []
```

scanl & scanr

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

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

scanr (#) v [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] -- ver prefissi
[[],[1],[2,1],[3,2,1]]
> scanr (:) [] [1,2,3] -- suffissi
[[1,2,3],[2,3],[3],[]]
> scanr (+) 0 [1,2,3] -- somma successivi
[6,5,3,0]
> scanl (+) 0 [1,2,3] -- somma precedenti
[0,1,3,6]
```

Derivare il codice di scanl: [ ]

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
```

dove **`inits`** sono i prefissi, cioè i segmenti iniziali di una lista (**Esercizio**). Si tratta di una definizione che porta a una comportamento **quadratico**. Tuttavia, possiamo fare delle trasformazioni. Partiamo dal **caso base**.

Da cui deriviamo che **`scanl`** è definita dall'equazione:

```
scanl f e [] = [e]
```

Caso []:

```
scanl f e []
   = { def di scanl }
map (foldl f e) . inits  []
   = { def di . }
map (foldl f e) (inits  [])
   = { def di inits }
map (foldl f e) [[]]
   = { def di map }
[foldl f e []]
   = { def di foldl }
[e]
```

Derivare il codice di scanl: x:xs

```
scanl f e (x:xs)
   = { def di scanl e .}
map (foldl f e) (inits  (x:xs))
   = { def di inits }
map (foldl f e) ([]: map (x:) init xs))
   = { def di map }
foldl f e [] : map (foldl f e) (map (x:) init xs))
   = { def di foldl }
e : map (foldl f e) (map (x:) init xs)
   = { map è un funtore }
e : map (foldl f e).(x:)) (init xs)
   = { occorre dimostrare (fold f e).(x:) = foldl f (f e x) }
e : map (foldl f (f e x)) (init xs)
   = { def. di scanl }
e : scanl f (f e x) xs
```

```
fold f e . (x:) xs
   = { def di (x:) }
(foldl f e) (x:xs)
   = { def di foldl }
foldl f (f e x) xs
```

Da cui (con il caso base) si derivano le equazioni per **scanl** di complessità **lineare**:

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

Lezione 8

That's all Folks…

Grazie per l'attenzione…

…Domande?