Tecniche di Programmazione Funzionale e Imperativa

Ivano Salvo

L'infinita importanza di Essere Lazy

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

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **11**, 29 marzo 2022

Lezione 11a:

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 di infiniti 1
ones = 1:ones

 -- 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]

 -- oppure…
> ones !! 111
1
```

Definizioni tipo & oggetti infiniti

Gli abitanti di un tipo ricorsivo in Haskell sono la **massima algebra chiusa rispetto** all'applicazione dei **costruttori**.

Ad esempio, i **numeri naturali** sono il **minimo insieme** chiuso rispetto al costruttore 0 e successore [Peano docet]:

"I numeri naturali sono 0 e tutti i successori di un numero naturale.

Nient'altro è un numero naturale"

La **co-algebra** (la massima algebra chiusa rispetto ai costruttori), contiene il "**naturale infinito**" $\omega = S\,(\,S\,(\,S\,(\,\ldots))) = S^{\infty}\,(\ldots)$

Vediamo se si possono fare computazioni interessanti…

```
data Nat = Z | S Nat
    deriving (Show, Eq, Ord)
-- definisco un naturale
-- infinito, omega
omega  = S omega
> tre < omega
True
> omega < tre
False
```

```
minNat = foldr min omega
minNat :: [Nat] -> Nat
> minNat []
S (S (S (S (S (S ( … ^C
> minNat [due, tre]
S (S Z)
> min due (minNat [])
S (S Z)
```

Per un ulteriore intuizione…

Lo stesso vale per le **liste**: la **minima algebra** chiusa rispetto a **(:)** e **[ ]** contiene **liste finite**.

Le **liste infinite** appaiono quando si considera la **massima algebra** chiusa rispetto ai costruttori.

Eliminando il costruttore costante **`Z`** da **`Nat`** (vedi definizioni del tipo **`NatInf`**) ottengo l'algebra che contiene **solo il "naturale" infinito** $\omega = S^{\infty}$

Analogamente, togliendo il costruttore lista vuota, e lasciando solo il costruttore cons **`C`** (vedi definizioni del tipo **`ListInf`**) ottengo l'algebra che contiene **solo stream, ossia liste infinite** costruite con infinite applicazioni di **`C`**.

```
data NatInf = S Nat
   deriving (Show, Eq, Ord)

data ListInf a = C a (ListInf a)
   deriving (Show, Eq, Ord)
```

Alcuni simpatici esempi…

Come definire lo **stream dei numeri naturali**? In Haskell possiamo indicarlo con lo zucchero sintattico: **`[0..]`**

Possiamo però **usare un generatore…**

Oppure aiutarci con i nostri funzionali sulle liste, in particolare **`map`**…ottenendo un risultato decisamene più elegante.

Analogamente possiamo fare le potenze di un numero, ad esempio… ci sono definizione migliori, dette **circolari**.

```
 -- 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)
 -- ci sono ottimi motivi per preferire:
powers' n = 1:ps where ps = 1:map (n*) ps
```

Problemi con liste infinite

Attenzione! non assumete che le funzioni **sappiano come sia fatta una lista infinita!**

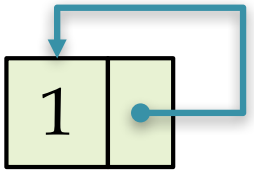
Il processo di generazione **deve consumare meno input di quanto output produce**!

Attenti anche a **list-comprehension** su liste infinite! o **filter**!

```
-- potremo essere tentati di definire:
factorsNT n = [x | x<-[1..], n `mod` x == 0 ]
> factorsNT 24
[1,2,3,4,6,8,12,24^CInterrupted.
-- non termina perché cerca in tutti i naturali...

-- meglio:
factors n = [x | x<-[1..n], n `mod` x == 0 ]
-- utile usare takeWhile che si ferma la
-- prima volta che la condizione è falsa
factors n = [x | x<-takeWhile (<n)[1..], n `mod` x == 0]
> factors 24
[1,2,3,4,6,8,12,24]

-- lo stream dei primi si definisce facilmente...
primes = [p | p<-[2..], factors p = [1,p]]
```

Definizioni Circolari

In una definizione come **ones** o **nats'**, il compilatore Haskell capisce che deve generare una lista infinita e durante la generazione è sufficiente **generare il nuovo primo elemento**.



Ricordate che una lista infinita, di fatto, è un processo…

Cosa che non gli è possibile per **powers** o **natGen** in cui l'occorrenza ricorsiva **dipende da un parametro**, tuttavia…

```
 -- Quindi è molto meglio:
powers' n = pws where pws = 1 : map (n*) pws

 -- Esempio funzione repeat:
repeat :: a -> [a]
myRepeatNaif x = x : myRepeatNaif x

myRepeat x = xs where xs = x:xs

 -- ad esempio:
map f = applyL (repeat f)
```

Definizioni Circolari: iterate

Facciamo un altro esempio e vediamo 3 versioni del funzionale predefinito iterate.

La prima definizione non crea una lista ciclica, ma **ogni riduzione produce un nuovo elemento della lista**: la computazione è **lineare nel numero di valori generati**.

La funzione **`iterate`**$_2$ è circolare, mentre **`iterate`**$_3$ è ottenuta dalla seconda eliminando la clausola where e quindi la dipendenza circolare.

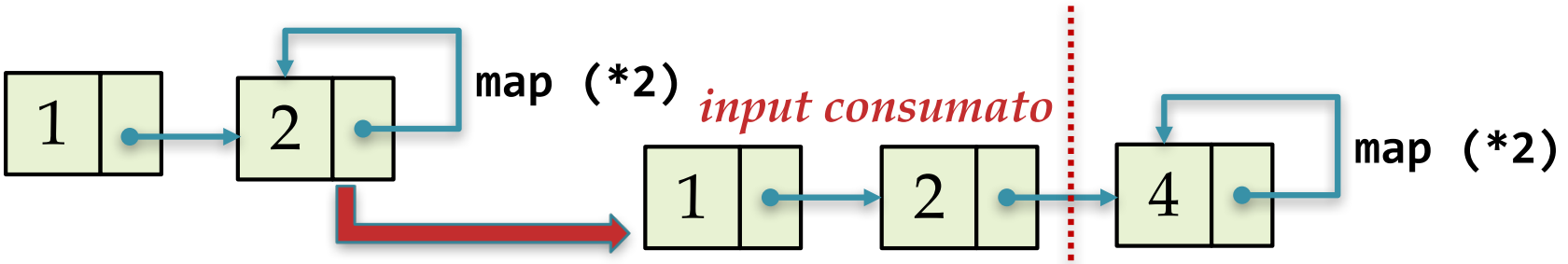
`iterate`$_2$ è anch'essa **lineare**, mentre **`iterate`**$_3$ è **quadratica**, e vediamo perché.

```
iterate :: (a -> a) -> a -> [a]
iterate1 f x = x : iterate1 f (f x)
iterate2 f x = xs where xs = x : map f xs
iterate3 f x = x : map f (f x)
```

Valutazione Lazy di Espressioni

Vediamo la differenza nella valutazione di **iterate$_2$** e **iterate$_3$**

Osservate che fs viene `prodotta` un elemento alla volta e il nostro processo **è sempre allineato all'ultimo elemento prodotto** quindi è sufficiente moltiplicare per 2 l'ultimo elemento generato!




```
iterate3 (*2) 1
→ 1 : map (*2) (iterate3 (*2) 1)
→ 1:2:map (*2) (map (*2) (iterate3 (*2) 1))
→ 1:2:4:map (*2) (map (*2)(map (*2) (iterate3 (*2) 1)))

iterate2 (*2) 1
→ 1 : map (*2) xs
→ 1 : 2 : map (*2) xs'
→ 1 : 2 : 4 : map (*2) xs''
```

produrre n elementi è quadratico in n

produrre n elementi è lineare in n

Una piccola chicca…

Definiamo lo stream dei numeri di **Fibonacci** usando un trucco noto: **due stream di fibonacci** un un passo avanti all'altro e sommo i numeri corrispondenti.

Visto che la definizione è **circolare**, in realtà esiste **un'unica copia** dello stream **`fibs`**.

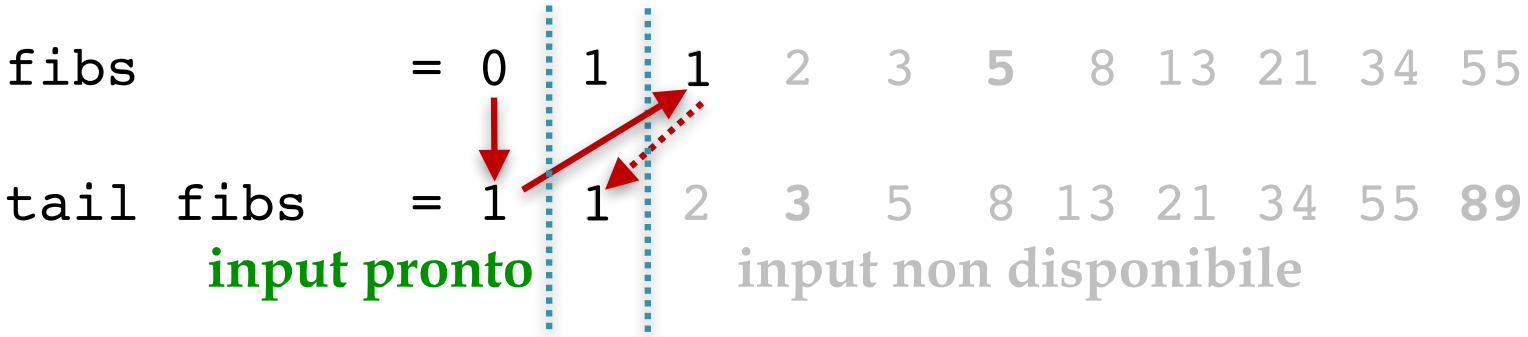
Le **definizioni guardate** danno una certa garanzia di **produttività**.

```
 -- Uso di un generatore (parametrico) per
 -- definire i naturali:
fibs = 0:1:zipWith (+) fibs (tail fibs)

 -- È più carino usare definizioni guardate,
 -- in cui non posso chiedere il tail e produco
 -- sempre almeno un elemento (ho mutua ricorsione)
fibs' = 0:fibs''
fibs'' = 1:zipWith (+) fibs' fibs''

 -- da cui, sostituendo in fibs''
 --(anche da fibs, spingendo dentro l'uno...)
fibs''' = 0:zipWith (+) fibs''' (1:fibs''')
```

Computazione di Fibonacci

Dovrebbe esservi naturale ragionare in termini di equazioni ricorsive, e trovare quella la **cui unica soluzione è la successione di Fibonacci**. Prendiamo un approccio più "artigianale".

Vediamo come procede la computazione: all'inizio **`fibs`** ha solo i primi due numeri, quindi è pronto anche il primo di **`tail fibs`**



Attenzione alle definizioni circolari! Il processo di generazione **deve consumare meno input di quanto output produce**!

L'equazione ricorsiva sotto **è soddisfatta da ogni stream** e infatti **non definisce niente**! È inconsistente!

```
 -- per esempio:
incstnts = (head incstnts):(tail incstnts)
```

Produttività

L'equazione nel riquadro non definisce nulla semplicemente perché dice qualcosa che **è vero per ogni lista o stream**.

Da un punto di vista computazionale, il problema è che lo stream **`incstnts`** non genera mai nessun elemento, mentre le funzioni **`head`** e **`tail`** lo vorrebbero consumare.

Si dice che la definizione **non è produttiva**: il processo di generazione circolare **si mangia la coda**, cioè **consuma più** input di **quanto ne produca**.

Questo fenomeno, come vedremo presto, può occorrere in forme più complicate e imprevedibile.

```
 -- stream inconsistente:
incstnts = (head incstnts):(tail incstnts)
```

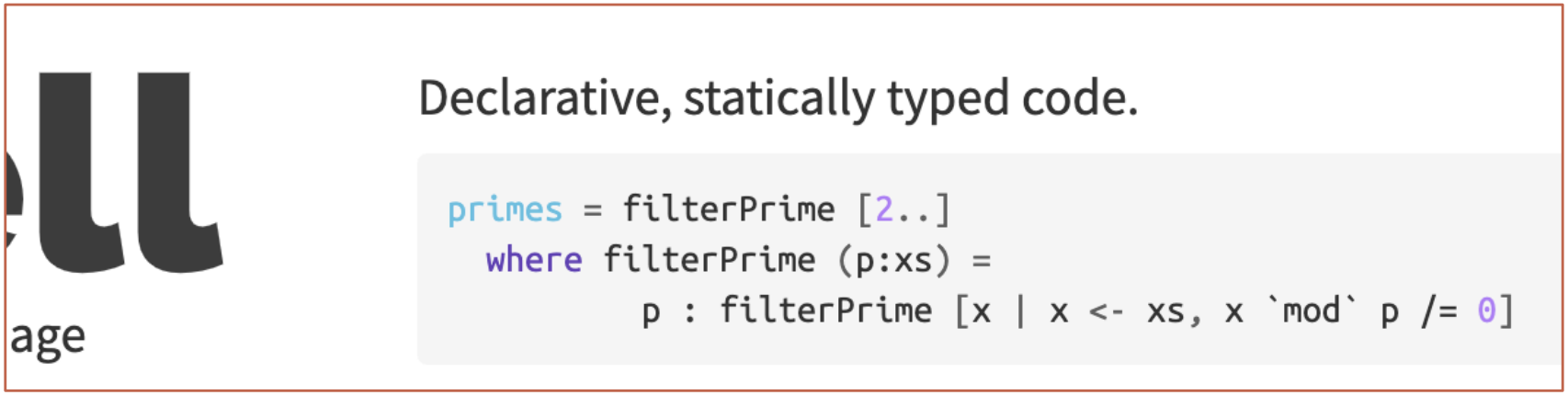
Primo assaggio (di primi)

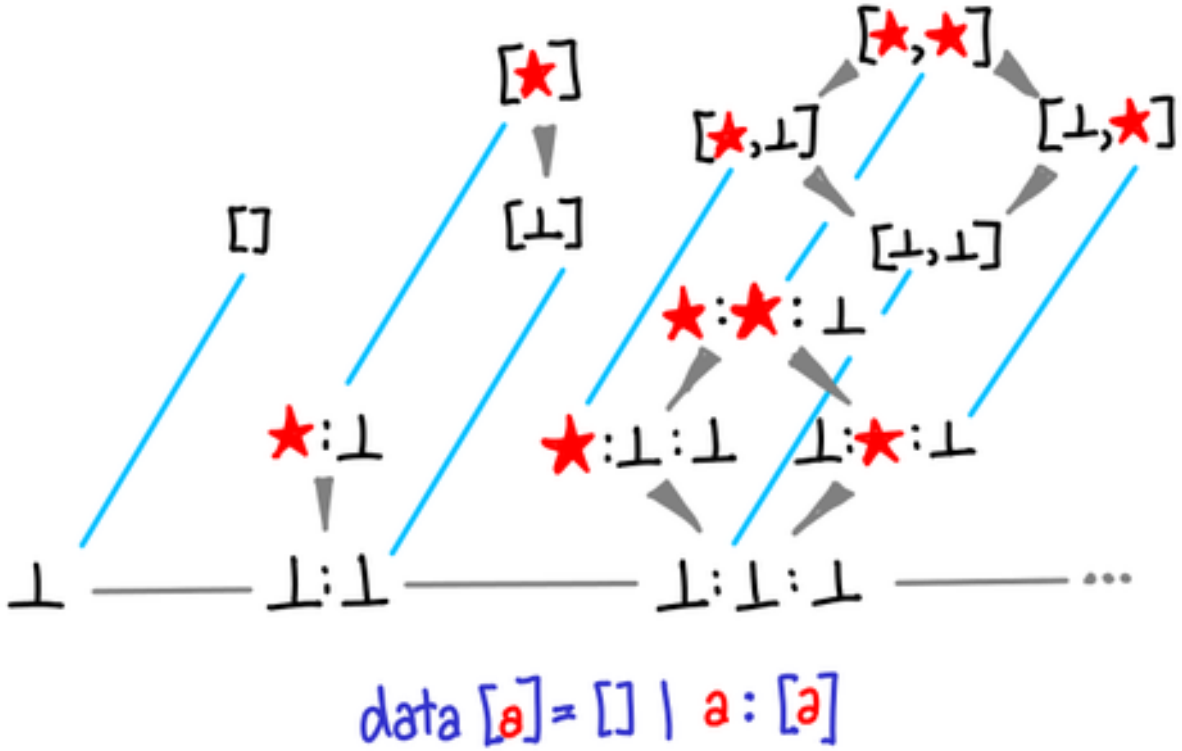
Da molti anni, sulla home page di **`www.haskell.org`**, c'è un programma che genera la **lista infinita dei numeri primi**.

Questo programma è stato pensato in epoca pre-Haskell da David Turner (1975), implementatore di diversi linguaggi funzionali, e ha **profondamente influenzato** scelte poi fatte in Haskell.

È noto come il *Crivello di Eratostene* in Haskell, anche se c'è accordo che di fatto non lo sia… in ogni caso l'idea è filtrare la lista di tutti i naturali eliminando tutti i numeri divisibili per il nuovo primo trovato…

… e funziona! (un po' lento, ma funziona…)





Lezione 11b:

Un po' di Teoria

Liste infinite come limiti

Una **lista infinita** può essere vista come il **limite** delle sue **approssimazioni finite**.

Per brevità indicheremo **`undefined`** con il simbolo ⊥ (= bottom).

Ad esempio la lista [1..] può essere vista come il limite della sequenza: `⊥, 1 :⊥, 1 : 2 :⊥, 1 : 2 : 3 :⊥, …` (anche di altre, a dire il vero).

Viceversa, la sequenza: `⊥,1 :⊥,2 : 1 :⊥,3 : 2 : 1 :⊥,…` **non converge** a **nessun limite**: il fatto è che la parte definita non si stabilizza a nessun valore e offre informazione contradditoria.

Osserviamo che le sequenze di approssimazioni possono convergere anche a una **lista finita**: è sufficiente che la parte non definita, a un certo punto diventi una lista finita, come ad esempio la lista vuota: `⊥,1 :⊥,1 : 2 :⊥,1 : 2 : [],… = [1, 2]`

La formalizzazione di questo concetto viene attraverso una nozione di **ordine parziale** ⊑ (ordin di approssimazione).

Approssimazioni: domini piatti

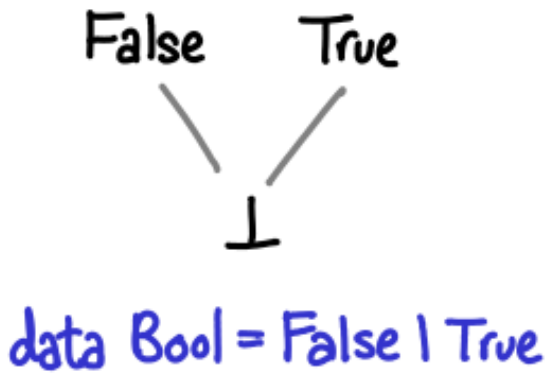
L'idea è che una **computazione incrementa** via vial'**informazione**.

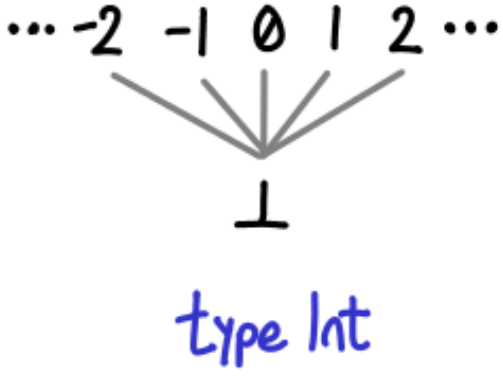
La cosa dovrebbe essere evidente proprio con le liste infinite: man mano che si calcolano, conosciamo più informazione della lista.

Ma ciò è vero in generale: pensate a un algoritmo di ordinamento: via via si avvicina alla soluzione, sistemando degli elementi.

In generale, per i tipi semplici, l'ordine è il cosiddetto **flat domain**: o un valore è calcolato, oppure è indefinito.
In formule $x \sqsubseteq y$ **sse** $x = \bot$ **oppure** $x = y$.

Aiutiamoci con le belle figure di Edward Z. Yang **nell'ezyang's blog** (`http://blog.ezyang.com/`).





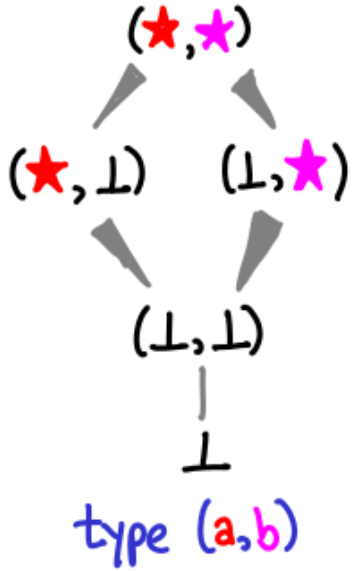
Approssimazioni: somme & prodotti

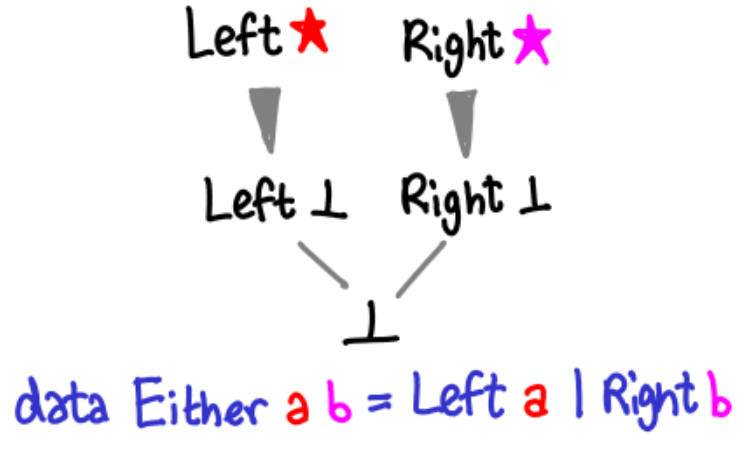
L'idea si estende ai **costruttori di tipo**.

Ad esempio, per le coppie, l'ordine è definito da: $(x, y) \sqsubseteq (x', y')$ sse $x \sqsubseteq x'$ e $y \sqsubseteq y'$.

Viceversa per **`Either`** (somme), si avrà $\bot \sqsubseteq$ **`Left`** $\bot$ e **`Right`** $\bot$ e a sua volta **`Left`** $x \sqsubseteq$ **`Left`** x' e **`Right`** $x \sqsubseteq$ **`Right`** x' sse $x \sqsubseteq x'$.

(nelle figure le **stelline** indicano "valore calcolato", e lì c'è tutta la complessità dell'ordine parziale dei tipi a e b).





Approssimazioni: liste & funzioni

L'ordine sulle **liste** è definito da:

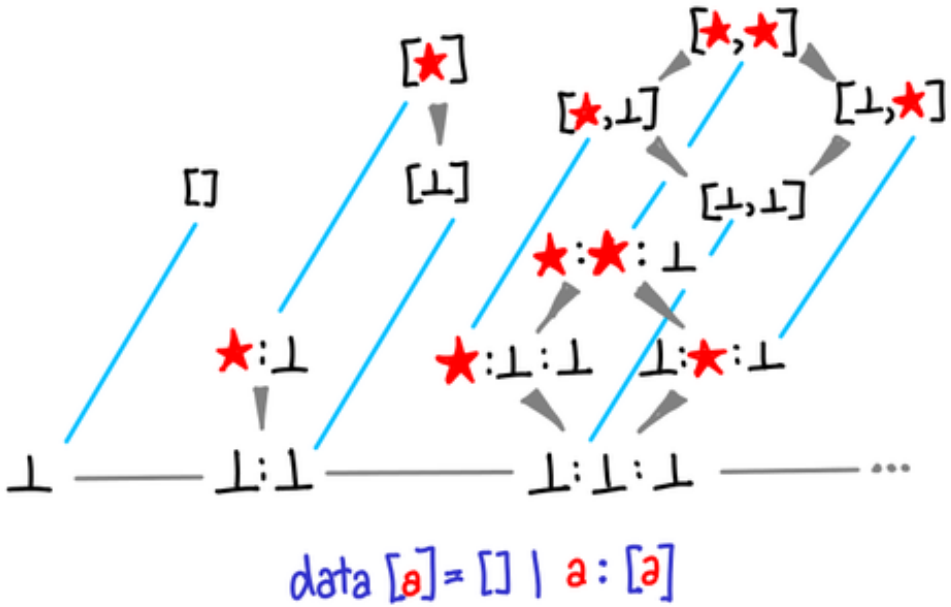
$$\bot \sqsubseteq xs$$

$$[\,] \sqsubseteq xs \text{ se e solo se } xs = [\,] \quad \text{e} \quad (x{:}xs) \not\sqsubseteq [\,]$$

$$(x{:}xs) \sqsubseteq (y{:}ys) \text{ se e solo se } x \sqsubseteq y \;\&\; xs \sqsubseteq ys$$

Ad esempio, abbiamo che `[1,⊥,3]` e `1: 2 :⊥` sono approssimazioni di `[1, 2, 3]`.

Per le **funzioni**, $f, g : a \to b$, $f \sqsubseteq_{a\to b} g$ se $\forall x.\, f\, x \sqsubseteq_b g\, x$.



Ordine, Limiti e Computabilità

Se ho una catena $x_1 \sqsubseteq x_2 \sqsubseteq x_3 \sqsubseteq \ldots$ in un certo tipo, necessariamente ho un limite $\sqcup x = \lim_{n\to\infty} x_n$ che soddisfa alle condizioni:

- per ogni n, $x_n \sqsubseteq \sqcup x$ (il **limite** è un **maggiorante**)
- se per ogni n, $x_n \sqsubseteq y$, allora $\sqcup x \sqsubseteq y$ (il **limite** è in effetti il **sup**)

Avendo queste proprietà, un ordine si dice **completo** (**CPO**, **C**omplete **P**artial **O**rder).

I tipi per domini di funzioni computabili, sono usualmente CPO.

Le **funzioni computabili** godono, di due importanti proprietà:

- **monotonia**: $x \sqsubseteq y$ allor $f\,x \sqsubseteq f\,y$.
- **continuità**: $f\,(\sqcup x) = \sqcup f\,(x)$

La **monotonìa** suggerisce che **più informazione ho nell'input**, **più** ne posso fornire in **output**.

La continuità è legata alla natura **finitaria** delle computazioni: una funzione computabile restituisce un valore **analizzando una porzione finita** di informazione.

Esempi di funzioni non computabili

- Consideriamo f, definita da $f(\bot)=0$ e $f(x)=1$ per $x \neq \bot$.

f non è computabile, perché f **non è monotona**. Necessariamente abbiamo $\bot \sqsubseteq x$, ma chiaramente $0 \not\sqsubseteq 1$.

Ne deduciamo che le uniche **funzioni non strette** su un **dominio flat** di un parametro sono le **costanti**.

- Consideriamo ora la funzione $g\ xs = \bot$ se xs è finita o parziale e $g\ xs = 1$ se xs è infinita.

g è monotona, ma **non è continua**. Quindi non computabile.

Infatti, g vale $\bot$ su **tutte le approssimazione finite** di una lista infinita, ma **1 sulla lista infinita** (limite).

Quindi $\bigsqcup g\ x_n = \bot \neq 1\ g(\bigsqcup x_n)$

Teoremi di punto fisso

Definizione: Un **reticolo completo** $(L, \sqsubseteq)$ è un insieme parzialmente ordinato in cui ogni insieme $A \subseteq L$ has un estremo inferiore $\sqcap A$ e un estremo superiore $\sqcup A$, e:

$$\sqcup A = min\{\, x \mid \forall\, a \in A.\ x \geq a \,\} \quad \text{e} \quad \sqcap A = max\,\{\, x \mid \forall\, a \in A.\ x \leq a \,\}$$

Teorema [KNARSTER-TARSKI]: In un reticolo $(L, \sqsubseteq)$, ogni funzione **monotona** ha un **massimo** e **minimo punto fisso**.

Teorema [KLEENE]: Se f **è continua**, il minimo punto fisso è $f^{\omega}(\bot)$, cioè $\lim_{n\to\infty} f^{n}(\bot)$.

Dimostrazione: Abbiamo visto che $\bot, f(\bot), f(f(\bot)), \ldots, f^{n}(\bot), \ldots$ è monotona crescente e quindi ha un sup. Da ciò:

$$\begin{aligned} f^{\omega} &= \sup\{f^{i} \mid i < \omega\} && \text{(def. of } f^{\omega}\text{)} \\ &= \sup\{f(f^{i}) \mid i < \omega\} && \text{(prestige ☺)} \\ &= f(\sup\{f^{i} \mid i < \omega\}) && \text{(continuity)} \\ &= f(f^{\omega}) && \text{(def. of } f^{\omega}\text{)} \quad \square \end{aligned}$$

Lezione 11

That's all Folks…

Grazie per l'attenzione…

…Domande?