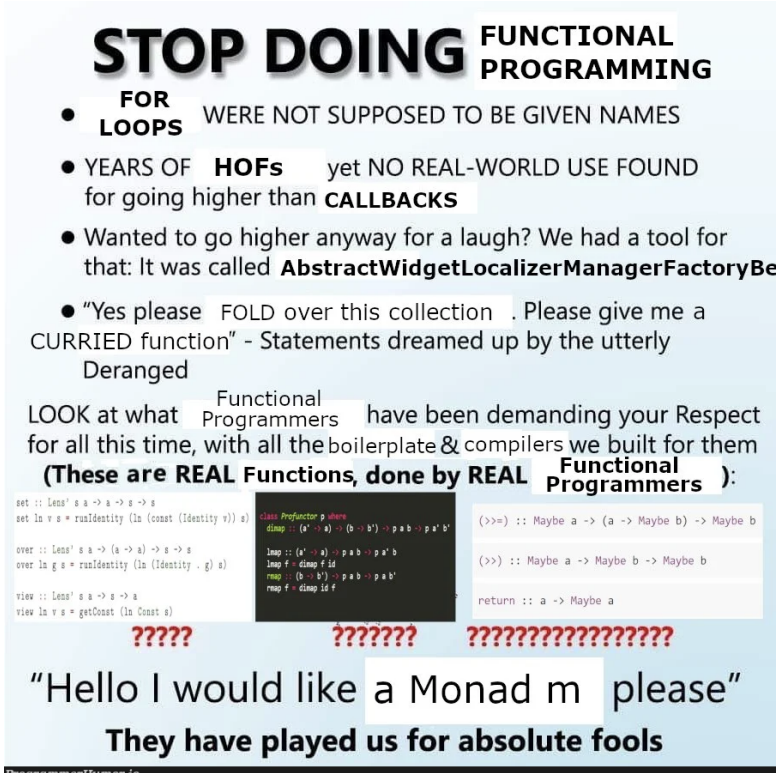
Sapienza Università di Roma

Tecniche di Programmazione Funzionale e Imperativa

libro del corso:
slide del professor Salvo



aglaia norza

2 maggio 2026

thisisaglaia@gmail.com
github.com/AglaiaNorza

Indice

1 Le basi **3**

1.1 Concetti fondamentali . . . 3

1.2 Sintassi, Funzioni e Pattern Matching . . . 4

1.2.1 Definizione e Applicazione . . . 4

1.2.2 Lambda Espressioni e Combinatori . . . 4

1.2.3 Pattern matching . . . 4

1.2.4 Scope locale: `let` e `where` . . . 5

1.3 Sistema dei Tipi e Classi . . . 6

1.3.1 Type Signatures . . . 6

1.3.2 Type Classes . . . 6

1.3.3 Definizione di Tipi: `type` e `data` . . . 7

1.3.4 Istanze e Derivazione (`deriving`) . . . 8

1.3.5 Tipabilità e Let-Polymorphism . . . 8

1.3.6 Semantica Denotazionale: ricorsione e punti fissi . . . 9

1.4 Strutture Dati Predefinite . . . 10

1.4.1 Tipi di Base e Tuple . . . 10

1.4.2 Liste . . . 10

1.5 Funzion(al)i di Ordine Superiore . . . 12

1.5.1 Famiglia dei `fold` (schemi di ricorsione) . . . 12

1.6 Alcuni esempi di stile funzionale . . . 13

2 Lambda Calcolo **15**

2.1 Sintassi e Computazione . . . 15

2.2 Variabili e Combinatori . . . 15

2.2.1 Combinatori noti e strutture di controllo 16
2.3 Ricorsione e Combinatori di Punto Fisso 16

3 Temi avanzati di programmazione funzionale 18
3.1 Fusion law per foldr 18
3.2 TODO: call-by-name + laziness (08) 20
3.3 TODO: funzioni strette e non strette 20
3.4 TODO: tecniche di ottimizzazione (09) 20
3.5 TODO: scanl e scanr (09bis) 20
3.6 Strutture dati infinite 20
3.6.1 Definizioni naif e circolari 20
3.7 Liste infinite come limiti 21
3.7.1 Domini, ordine e limiti 22
3.8 Teoremi di punto fisso 24
3.9 TODO: induzione e coinduzione 26
3.10 Funtori e Applicativi 26
3.10.1 Funtori 26
3.10.2 Applicativi 28
3.11 TODO: monadi (14) 30
3.12 TODO: monadi I/O ecc (15) 30

1 Le basi

1.1 Concetti fondamentali

- **Valutazione lazy:** Haskell valuta le espressioni solo quando è strettamente necessario (permette di gestire agevolmente strutture dati infinite)
- **Trasparenza referenziale:** non essendoci assegnazioni o memoria mutabile (assenza di side-effects), una variabile identifica sempre lo stesso oggetto (permette di manipolare algebricamente i programmi)
- **Polimorfismo e type inference:** molte funzioni operano su tipi generici (es. `a`, `b`). Il compilatore inferisce automaticamente il *tipo principale* (il più generale possibile); tutti gli altri tipi corretti sono sue istanze.
- **Non-terminazione, Bottom e `undefined`:** in Haskell, ogni tipo possiede implicitamente un valore speciale (chiamato *bottom*, indicato matematicamente con $\bot$) che rappresenta un'eccezione fatale o una computazione che entra in un loop infinito
  - **la costante `undefined`:** è la materializzazione a livello di codice del valore $\bot$ - poiché una computazione che fallisce o non termina può avvenire in qualsiasi contesto (mentre si aspetta un numero, un booleano, ecc.), `undefined` possiede il tipo più generale e polimorfo possibile: `a` (ovvero $\forall\alpha.\alpha$)
  - **Il limite del lambda calcolo (*Occurs check*):** nel lambda calcolo puro, il loop infinito per eccellenza è il combinatore $\Omega = (\lambda x.xx)(\lambda x.xx)$. In Haskell, l'auto-applicazione $\lambda x.xx$ **non è tipabile**. Perché x possa prendere se stesso come argomento, il suo tipo α dovrebbe essere uguale a una funzione che prende α come parametro e restituisce β ($\alpha = \alpha \to \beta$). Questo creerebbe un tipo infinito (durante l'inferenza dei tipi, il compilatore esegue un controllo chiamato *Occurs check* che impedisce alla variabile di tipo α di apparire all'interno della sua stessa definizione, rifiutando di conseguenza l'espressione)
  - **Ricorsione esplicita:** poiché il type system ci vieta di creare loop tramite l'auto-applicazione anonima di $\lambda x.xx$, in Haskell la non-terminazione si esprime unicamente attraverso la ricorsione esplicita. Definendo ad esempio `omega' = omega'`, creiamo intenzionalmente una computazione divergente che il compilatore accetta, assegnandole il tipo più generale `a`.
- **Ideologia funzionale:** l'ideologia funzionale favorisce l'assemblaggio di funzioni semplici tramite composizione (`.`). Si pensa in termini di trasformazioni globali di strutture dati, per poi raffinare in un secondo momento i colli di bottiglia prestazionali.

1.2 Sintassi, Funzioni e Pattern Matching

1.2.1 Definizione e Applicazione

- l'applicazione di funzione: **associa a sinistra** e si scrive senza virgole (`f x y`).
- **aperatori infissi vs prefissi:** le funzioni binarie si usano in modo infisso tra backtick (`10 ‘div‘ 2`), gli operatori infissi in modo prefisso tra parentesi (`(+) 3 4`).

1.2.2 Lambda Espressioni e Combinatori

Le funzioni anonime si scrivono con backslash (`\` simula λ).

```haskell
-- identita' (combinatore I)
i :: t -> t
i x = x           -- definizione standard
i' = \x -> x      -- lambda espressione

-- combinatore K (proiettore del primo argomento) (primo assioma di Hilbert)
k :: t1 -> t2 -> t1
k' = \x y -> x

-- combinatore S (sostituzione) (secondo assioma di Hilbert)
s :: (t2 -> t1 -> t) -> (t2 -> t1) -> t2 -> t
s x y z = x z (y z)
```

1.2.3 Pattern matching

Il pattern matching decompone le strutture dati in base alla loro forma sintattica. L'ordine delle clausole è strettamente sequenziale (viene scelta la prima dall'alto che fa matching).

- **mancanza di unificazione:** non si possono usare variabili identiche nel pattern per verificarne l'uguaglianza (es. `f x x = ...` è invalido)
- **underscore (`_`):** variabile anonima ("don't care") per valori non rilevanti
- **as-pattern (`@`):** permette di dare un nome all'intera struttura mentre la si decompone (es. `xs@(x:txs)`)
- **guardie (`|`):** permettono di definire il comportamento di una funzione valutando sequenzialmente una serie di *predicati booleani* (condizioni logiche); a differenza del pattern matching che controlla la *forma* sintattica dei dati, le guardie valutano *valori e relazioni* (es. `x > 0`, `x == y`)
  - l'ultima guardia è tipicamente `otherwise` (che nel Prelude di Haskell è semplicemente un alias per `True`), che funge da caso di default ("catch-all") per assicurarsi che la funzione restituisca sempre un valore.

```haskell
confronta x y
  | x == y    = "i numeri sono uguali" -- supera la mancanza di unificazione
  | x > y     = "il primo e' maggiore"
  | otherwise = "il secondo e' maggiore"
```

- **equazioni multiple (definizione per casi):** il modo più comune di fare pattern matching è definire la funzione scrivendo più "intestazioni" separate; il compilatore le prova dall'alto verso

il basso: se gli argomenti combaciano con la forma (pattern) della riga, esegue quell'equazione, altrimenti controlla la successiva.

```
myAnd True True = True
myAnd   _    _  = False
```

– è meno comune, ma è possibile fare pattern-matching anche nelle *lambda-espressioni*:

```
myFst' = \(x, y) -> x
```

1.2.4 Scope locale: `let` e `where`

- **`let ... in`**: costrutto standalone per definire variabili/funzioni locali (utile per strutturare/destrutturare tuple nelle ricorsioni)
- **`where`**: clausola che qualifica la parte destra di una definizione appena conclusa

```
fibAux n = let (f, fPrec) = fibAux (n-1) in (f + fPrec, f)
-- equivalente a:
fibAux' n = (f + fPrec, f) where (f, fPrec) = fibAux' (n-1)
```

> differenza tra `let` e `where`
>
> una clausola `where` è visibile da tutte le guardie di una specifica equazione. Un `let` definito dopo il simbolo `=` di una guardia è visibile solo all'interno di quella singola guardia. Per condividere un calcolo tra più guardie, il `where` è obbligatorio.

```
-- condivisione tra guardie
analizzaRaggio r
  | area > 100 = "cerchio grande"
  | area < 10  = "cerchio piccolo"
  | otherwise  = "cerchio medio"
  where area = pi * r^2  -- 'area' calcolata una volta e visibile a tutti i pipe

-- inline: (serve il let perche' non c'e' una dichiarazione di funzione)
volumeCilindro r h = (let area = pi * r^2 in area) * h
```

1.3 Sistema dei Tipi e Classi

Haskell è **statically type-safe**: se un programma è tipato correttamente, nessun errore di tipo si verifica durante l'esecuzione.

1.3.1 Type Signatures

In Haskell, grazie al meccanismo di type inference, il compilatore è in grado di dedurre automaticamente il tipo più generale per quasi ogni espressione. Tuttavia, è considerata un'ottima pratica (e in alcuni casi più complessi è strettamente necessario) dichiarare esplicitamente il tipo di una funzione subito sopra la sua definizione.

Questo si fa usando l'operatore `::` (che si legge "ha tipo"). Dichiarare esplicitamente i tipi è utile ai fini della leggibilità del codice e permette di "forzare" il tipo desiderato, aiutando il compilatore a individuare eventuali errori logici in modo molto più preciso.

```haskell
-- dichiarazione esplicita del tipo: prende due Int e restituisce un Int
somma :: Int -> Int -> Int
somma x y = x + y
```

1.3.2 Type Classes

Le classi limitano il polimorfismo imponendo vincoli (o "interfacce") sui tipi. Più che semplici insiemi, rappresentano classificazioni di tipo algebrico: raggruppano i tipi su cui sono definite certe operazioni. Le classi possono formare gerarchie, dove una sottoclasse "estende" la superclasse ereditandone le operazioni e aggiungendone di nuove.

- **Eq (Equality Types):** comprende i tipi che ammettono l'uguaglianza (`==`) e la sua negazione (`/=`). Tutti i tipi base (Bool, Int, Float, ecc.) sono istanze di Eq, così come le liste e tuple costruite su tipi Eq

  definizione interna di Eq

  ```haskell
  class Eq a where
    (==), (/=) :: a -> a -> Bool
    -- le classi possono contenere codice (metodi di default)
    x /= y = not (x == y)
  ```

- **Ord (Ordered Types):** sottoclasse di `Eq` - tipi ordinabili che supportano operazioni come `<=` , `<` , `>=` , `>` , `min` e `max` .
  - per definire un'istanza di Ord, è sufficiente fornire il codice per l'operatore `<` , il resto può essere interdefinito.
- **Num:** la classe più generica per i tipi numerici. È sottoclasse di `Eq` e `Show` . Implementa operazioni aritmetiche base (`+` , `-` , `*`), oltre a valore assoluto (`abs`) e segno (`signum`).

  i numeri interi costanti sono polimorfi rispetto alla classe `Num` :

  ```haskell
  > :t 3
  3 :: Num a => a  -- 3 puo' essere un Int, Integer, Float, ma non un Char
  ```

- **Altre classi utili:**

- **Show:** tipi che possono essere stampati (convertiti in `String`) tramite il metodo `show`. Se provate a stampare a video il risultato di una funzione che non appartiene a `Show` (es. una lambda anonima come `\x -> x`), l'interprete darà errore.
- **Read:** permette di trasformare una `String` in un valore tramite il metodo `read`
- **Integral:** sottoclasse di `Num` per i numeri interi (es. Int, Integer); aggiunge le operazioni di divisione intera `div` e modulo `mod`
- **Fractional:** sottoclasse di `Num` per i numeri frazionari (es. Float, Rational); aggiunge l'operatore di divisione reale `(/)` e `recip`

1.3.2.1 Vincoli di classe

I vincoli di classe si indicano prima del tipo vero e proprio, separati dalla doppia freccia `=>`. Vincoli multipli si racchiudono tra parentesi e indicano che la funzione opera su un tipo che deve soddisfare tutte le classi elencate.

```haskell
-- esempio: mcd richiede che il tipo 'a' supporti sia l'ordine (<, ==)
-- sia le operazioni aritmetiche (-)
mcd :: (Ord a, Num a) => a -> a -> a
mcd x y
  | x == y    = x
  | x < y     = mcd (y - x) x
  | otherwise = mcd (x - y) y
```

1.3.3 Definizione di Tipi: `type` e `data`

- **`type` (sinonimi):** danno nuovi nomi a tipi esistenti (es. `type Casella = (Int, Int)`)
- **`data` (tipi algebrici):** costruiscono nuovi tipi tramite costruttori finiti, parametrici o ricorsivi. Possono essere dichiarati istanze di classi.

```haskell
data SetteNani = Eolo | Pisolo | Brontolo          -- finiti
data Either a b = Left a | Right b                 -- unioni disgiunte
data Maybe a = Just a | Nothing                    -- gestione fallimenti
data List a = Nil | Cons a (List a)                -- ricorsivi/induttivi
data BTree a = Foglia a | Radice (BTree a) (BTree a)
```

1.3.3.1 Il costruttore di tipo `Maybe`

Il tipo `Maybe` è il tipo di computazioni che possono fallire. In Haskell, è il tipo con cui si rappresentano comunemente le eccezioni in maniera sicura e controllata dal sistema di tipi (corrisponde a `optional`)

Un valore di tipo `Maybe` può assumere due forme:

- `Just v`: contiene un valore `v` (la computazione ha avuto successo e restituisce qualcosa)
- `Nothing`: non contiene nulla (rappresenta una computazione indefinita o fallita)

Si può utilizzare per le *funzioni parziali* (che non sono definite per tutti i possibili valori di input (eg divisione per zero)). Tramite il pattern matching, possiamo "spacchettare" il dato per estrarne il contenuto.

```haskell
-- definizione del tipo predefinito
data Maybe a = Just a | Nothing

-- 1. costruzione di un Maybe: divisione sicura
-- Se il divisore e' 0 restituisce Nothing, altrimenti restituisce Just il
    risultato
safediv :: Integral a => a -> a -> Maybe a
safediv n 0 = Nothing
safediv n m = Just (n `div` m)

-- 2. destrutturare un Maybe: funzione "inversa" per estrarre il valore
extract :: Maybe a -> a
extract (Just n) = n
extract Nothing  = undefined

-- extract (safediv 30 0) -> *** Exception: Prelude.undefined
```

1.3.4 Istanze e Derivazione (`deriving`)

La keyword `deriving` è un meccanismo automatico che permette al compilatore di generare le istanze per le classi predefinite senza dover scrivere codice manuale.

- **Eq:** genera un'uguaglianza basata sulla sintassi
- **Ord:** deriva l'ordine lessicografico in base alla sequenza dei costruttori
- **Show/Read:** la conversione usa i nomi dei costruttori come stringhe
- **Enum:** permette di generare liste per enumerazione (es. `[Lun..Ven]`)

```haskell
-- istanze automatiche tramite deriving
data Giorni = Lun | Mar | Mer | Gio | Ven | Sab | Dom
    deriving (Eq, Ord, Show, Enum, Read)

-- esempio di istanza manuale
instance Eq Bool where
    False == False = True
    True == True   = True
    _ == _         = False
```

1.3.5 Tipabilità e Let-Polymorphism

- Non tutto è tipabile in Haskell. Il classico esempio è l'auto-applicazione pura:
  - `omega = \x -> x x` non è tipabile perché richiederebbe un tipo infinito ($t = t \to t_1$), portando a una teoria dei tipi non decidibile (e quindi a tipi non inferibili dal compilatore)
- Dal punto di vista del calcolo, `let x = N in M` è equivalente all'applicazione di una lambda: `(\x -> M) N`. Tuttavia, il compilatore Haskell tratta i due casi in modo diverso durante l'inferenza dei tipi.
- **Let-Polymorphism:**

  Il costrutto `let` permette di assegnare a una variabile un tipo polimorfo che viene "generalizzato" prima di essere usato nel corpo (`in`).
  - **tipabile:** `let x = \y -> y in x x` è tipabile.

Haskell vede che `x` è l'identità ($a \to a$) e, nel corpo `x x`, può istanziare il primo `x` come $(a \to a) \to (a \to a)$ e il secondo come $(a \to a)$.

– **non tipabile:** `(\x -> x x) (\y -> y)` **non** è tipabile.
Qui il compilatore deve tipare la funzione `\x -> x x` prima di sapere che `x` sarà l'identità. Come visto sopra, `\x -> x x` è intrinsecamente non tipabile.

> Lemma 1: **Terminazione**
>
> Tutti i lambda-termini tipabili nel sistema dei tipi di Haskell (che non usino ricorsione esplicita tramite equazioni) **terminano**.

1.3.6 Semantica Denotazionale: ricorsione e punti fissi

Haskell è un linguaggio dichiarativo: una definizione non è un comando, ma un'equazione. Quando definiamo una funzione ricorsiva f, stiamo scrivendo un'equazione del tipo $f = \mathcal{E}(f)$, dove $\mathcal{E}$ è un'espressione che contiene f stessa.

- **Bottom ($\perp$)** rappresenta il "non-valore". Indica un calcolo che non termina (loop infinito) o che fallisce (errore). In Haskell, ogni tipo t contiene implicitamente $\perp$.
- Se la definizione è un'equazione, la "semantica" (ovvero la funzione reale prodotta dal compilatore) deve essere una soluzione dell'equazione. Matematicamente, una soluzione di $f = T(f)$ è un **punto fisso** del funzionale T.

> Approfondimento: ricorsione e non-terminazione: `omega'`
>
> Prendiamo la funzione

```haskell
omega' x = omega' x
-- > :t omega'
-- omega' :: t -> t1
```

> Questa funzione ha il tipo più generico possibile (`t -> t1`) perché, non terminando mai, non produce mai un valore reale.
>
> Rimuovendo x, otteniamo $omega' = I(omega')$, dove I è l'identità.
>
> Qualsiasi funzione f soddisfa $f = I(f)$. Quale scegliere?
>
> - Haskell sceglie la funzione che "fa il minimo indispensabile" per soddisfare l'equazione. Si usa l'ordinamento $\sqsubseteq$ dove $f \sqsubseteq g$ significa che g è "più definita" di f.
> - **Costruzione di Kleene (least-fixpoint):**
>
> il computer "costruisce" la funzione partendo dal nulla ($\perp$):
>
> $$f_0 = \perp \quad \text{(non so nulla)}$$
>
> $$f_1 = T(f_0) \quad \text{(applico la definizione una volta)}$$
>
> $$f_\infty = \text{limite della catena} = \mathit{lfp}(T)$$
>
> Nel caso di `omega'` , la catena è $\{\perp, \perp, \dots\}$. Il limite è $\perp$. La semantica di `omega'` è dunque la funzione che restituisce sempre "non-valore".

Il punto fisso è quindi la "semantica" perché è l'unico modo per dare un valore univoco e calcolabile a una definizione circolare. Il *minimo* punto fisso garantisce che il significato sia quello "prodotto" effettivamente dal calcolo ricorsivo.

`myundefined`

Mentre `omega'` è una *funzione* che non termina, possiamo definire un *valore* che non termina:

```
myUndefined = myUndefined
-- > :t myUndefined
-- myUndefined :: a
```

- Poiché `myUndefined` ha tipo `a`, esso **appartiene a tutti i tipi**. È l'implementazione "home-made" della costante `undefined` di Haskell. Rappresenta il valore $\perp$ per ogni possibile tipo del linguaggio.
- Matematicamente, `myUndefined` è il **punto fisso dell'identità**:
  (1) la definizione `x = x` corrisponde all'equazione funzionale $x = I(x)$
  (2) la soluzione di questa equazione è, per definizione, il punto fisso dell'identità I
  (3) poiché Haskell usa la semantica del minimo punto fisso, e l'identità applicata al valore nullo ($\perp$) restituisce ancora $\perp$, il risultato è la non-terminazione
  (4) questo spiega perché `myUndefined` ha tipo `a`: non essendo "sceso" verso nessun valore specifico (come un intero o una stringa), rimane una pura astrazione di errore/loop che può "fingere" di essere qualsiasi tipo.

Haskell permette quindi di manipolare la non-terminazione come un valore reale. Ogni tipo in Haskell è in realtà "puntato" (*pointed*), ovvero contiene sempre almeno un valore: $\perp$.

1.4 Strutture Dati Predefinite

1.4.1 Tipi di Base e Tuple

- **Booleani:** `Bool` (`True`, `False`). Operatori: `not`, `&&`, `||`.
- **Interi:** `Int` (dimensione fissa), `Integer` (precisione illimitata).
- **Tuple:** n-uple fisse e disomogenee (es. `(String, Bool)`)
  offrono le funzioni *su coppie*: `fst`, `snd` (prendono primo e secondo argomento)

1.4.2 Liste

Sequenze di elementi omogenei. Definite dal costruttore vuoto `[]` e dall'operatore infisso *cons* `(:)`. Le stringhe sono `[Char]`.

Zucchero sintattico: `[1,2,3]` equivale a `1:2:3:[]`.

```
-- enumerazioni
[m..n]          -- range finito (lista con elementi da m a n)
[m, n..p]       -- range con step (es. [1, 3..11] dispari)
```

```haskell
[m..]           -- lista infinita (lazy)
```

Haskell supporta le **list comprehension** (definizioni set-like del contenuto di una lista)

```haskell
-- list comprehension (spesso tradotte internamente in map/filter)
[x^2 | x <- [1..5]]              -- quadrati
[x | x <- xs, x `mod` 2 == 0]    -- filtri / guardie
[(x,y) | x <- xs, y <- ys]       -- prodotto cartesiano
```

1.4.2.1 Funzioni su Liste

- **Estrazione:** `head`, `tail`. `last`, `init`. `(!!)` per estrarre all'indice n

```haskell
head [1, 2, 3]  -- risultato: 1
tail [1, 2, 3]  -- risultato: [2, 3]
last [1, 2, 3]  -- risultato: 3
init [1, 2, 3]  -- risultato: [1, 2]
[1, 2, 3] !! 1  -- risultato: 2
```

- **Ispezione:** `null`, `length`

```haskell
null []          -- risultato: True
length [1, 2, 3] -- risultato: 3
```

- **Taglio:** `take n`, `drop n`, `splitAt n` (tupla con take e drop).

```haskell
take 2 [1, 2, 3, 4]    -- risultato: [1, 2]
drop 2 [1, 2, 3, 4]    -- risultato: [3, 4]
splitAt 2 [1, 2, 3, 4] -- risultato: ([1, 2], [3, 4])
```

- **Generali:** `++` (concatenazione), `reverse`

```haskell
[1, 2] ++ [3, 4]  -- risultato: [1, 2, 3, 4]
reverse [1, 2, 3] -- risultato: [3, 2, 1]
```

- **Matematica/logica:** `sum`, `minimum`, `maximum`. `and`, `or`. `any p`, `all p` (predicati su almeno uno o tutti)

```haskell
sum [1, 2, 3]       -- risultato: 6
minimum [5, 2, 9]   -- risultato: 2
maximum [5, 2, 9]   -- risultato: 9
and [True, False]   -- risultato: False
or [True, False]    -- risultato: True
any (> 2) [1, 2, 3] -- risultato: True (almeno uno è > 2)
all (> 0) [1, 2, 3] -- risultato: True (tutti sono > 0)
```

- **Liste annidate/multiple:** `concat` (appiattisce liste di liste), `zip` (unisce liste in coppie, fermandosi alla più corta)

```haskell
concat [[1, 2], [3, 4]]    -- risultato: [1, 2, 3, 4]
zip [1, 2] ['a', 'b', 'c'] -- risultato: [(1, 'a'), (2, 'b')]
```

1.5 Funzion(al)i di Ordine Superiore

I funzionali sono funzioni che prendono altre funzioni come argomento o le restituiscono come risultato. Favoriscono la **composizionalità**, permettendo di costruire programmi complessi tramite piccoli blocchi riutilizzabili.

- **curryficazione:** si fonda sull'isomorfismo $A \times B \to C \cong A \to (B \to C)$. Le versioni curryficate permettono di passare un solo argomento alla volta, favorendo l'applicazione parziale.
- **η-regola:** è possibile omettere un parametro in una definizione se esso compare identico alla fine di entrambi i lati dell'equazione.

```haskell
-- composizione (.) : Equivale a f(g(x))
(.) f g x = f (g x)

-- curry e uncurry (Isomorfismo A x B -> C  < = = >  A -> (B -> C))
curry :: ((a, b) -> c) -> a -> b -> c
uncurry :: (a -> b -> c) -> (a, b) -> c

-- flip: inverte l'ordine dei parametri
flip :: (a -> b -> c) -> b -> a -> c
myFlip f a b = f b a

-- flip (:) [2, 3] 1
-- [1, 2, 3]

-- map e filter
map :: (a -> b) -> [a] -> [b]          -- proprieta': map (f . g) = map f . map g
filter :: (a -> Bool) -> [a] -> [a]

-- zipWith: applica f. binaria agli elementi di due liste
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

1.5.1 Famiglia dei `fold` (schemi di ricorsione)

I `fold` astraggono il concetto di ricorsione strutturale sulle liste, sostituendo i costruttori della struttura dati (l'operatore `(:)` e la lista vuota `[]`) con funzioni e valori specifici. Se l'operatore `#` è associativo e ha `e` come elemento neutro, allora `foldl (#) e` $=$ `foldr (#) e`.

- **`foldr` (right fold):** associa a destra; la riduzione avviene "al ritorno" dalle chiamate ricorsive (costruisce espressioni).
  - sostituisce ricorsivamente il costruttore `(:)` con la funzione `f` data e `[]` con il valore base `z`.
  - espansione: `foldr f z [x1, x2, x3]` diventa `x1 ‘f‘ (x2 ‘f‘ (x3 ‘f‘ z))`
  - può terminare anche su liste infinite se la funzione `f` non è stretta nel suo secondo argomento (ovvero se non ha sempre bisogno di valutare la ricorsione per restituire un risultato).

```haskell
-- somma con foldr
foldr (+) 0 [1, 2, 3]
-- valutazione: 1 + (2 + (3 + 0)) = 6

-- implementazione di map usando foldr
myMap f xs = foldr (\x acc -> f x : acc) [] xs
```

- **`foldl` (left fold):** associa a sinistra; simula un ciclo iterativo passando un accumulatore "in avanti"
  - valuta l'espressione partendo da sinistra e procedendo verso l'interno - valuta tutto solo alla fine della lista
  - espansione: `foldl f z [x1, x2, x3]` diventa `((z 'f' x1) 'f' x2) 'f' x3`
  - *nota bene:* In `foldl`, la funzione prende come primo argomento l'accumulatore, e come secondo l'elemento corrente della lista.

```haskell
-- sottrazione con foldl
foldl (-) 0 [1, 2, 3]
-- valutazione: ((0 - 1) - 2) - 3 = -6

-- sottrazione con foldr per confronto
foldr (-) 0 [1, 2, 3]
-- valutazione: 1 - (2 - (3 - 0)) = 2

-- reverse usando foldl
myReverse xs = foldl (\acc x -> x : acc) [] xs
```

- **`foldr1` e `foldl1`:** varianti che non richiedono un valore iniziale esplicito, poiché usano il primo o l'ultimo elemento come base; *danno errore su lista vuota*.

```haskell
-- utile quando non esiste un elemento neutro sensato (es. min/max assoluto)
myMinimum = foldr1 min
myMinimum [5, 3, 8] -- risultato: 3
-- myMinimum []     -- genera errore a runtime
```

1.6 Alcuni esempi di stile funzionale

Lo stile funzionale evita accumulatori ed indici, preferendo la manipolazione globale delle strutture dati.

- **prodotto scalare:** può essere implementato in diversi modi (flessibilità dei funzionali):

```haskell
-- accoppia gli elementi, trasforma "*" in una funzione che accetta una tupla
-- applica la moltiplicazione agli elementi, e somma i prodotti
ps1 xs ys = sum (map (uncurry (*)) (zip xs ys))

-- crea una lista di funzioni parzialmente applicate
-- (da [5, 4] si ottiene [(5*), (4*)])
-- zApp/applyL prende la lista di funzioni ^ e la applica sulla seconda lista
ps2 xs ys = sum (applyL (map (*) xs) ys)

-- zipWith fa zip e map in un passo solo
```

```haskell
ps3 xs ys = sum (zipWith (*) xs ys)
```

- controllare se una lista è ordinata:

```haskell
ordinata xs = and (zipWith (<=) xs (tail xs))
```

- **ricerca su lista** (`findVPH`): si può implementare accoppiando la lista agli indici (`zip [0..] xs`) e filtrando per il valore cercato.

2

Lambda Calcolo

Ideato da Alonzo Church nel 1931, il λ-calcolo nasce per esplicitare il *processo meccanico di calcolo* (sintassi) di una funzione, in contrasto con la visione matematica classica che vede la funzione solo come una relazione statica tra insiemi (semantica).

2.1 Sintassi e Computazione

La sintassi del λ-calcolo è minimalista ed è composta da tre regole:

- **variabile:** $x, y, z \dots$
- **astrazione:** $\lambda x.M$ (corrisponde alla definizione di una funzione con parametro x e corpo M)
- **applicazione:** (MN) (corrisponde all'applicazione della funzione M all'argomento N)

in BNF, quindi, la grammatica del lambda-calcolo è quindi:

$$T ::= V \mid \lambda V.T \mid (T\ T)$$

> Regole di associazione
>
> - **l'applicazione associa a sinistra:** $FN_1N_2 \dots N_n$ equivale a $(\dots(FN_1)\dots N_n)$
> - **l'astrazione associa a destra:** $\lambda x_1 \dots x_n.M$ equivale a $\lambda x_1.(\lambda x_2.(\dots(\lambda x_n.M)\dots))$

La computazione avviene tramite la β**-riduzione** (beta-regola): un termine della forma $(\lambda x.M)N$ si dice *redex* e si riduce sostituendo tutte le occorrenze di x in M con il termine N. Formalmente:

$$(\lambda x.M)N \to_\beta M[N/x]$$

Un termine che non contiene più β-redex è detto in **forma normale** e rappresenta un valore finale (il risultato della computazione).

2.2 Variabili e Combinatori

- **Variabili libere (FV) e legate:** in $\lambda x.M$, la variabile x è *legata* (bound) all'astrazione. Qualsiasi variabile non legata è *libera* (free).
- α**-conversione (alfa-regola):** per evitare la cattura accidentale di variabili libere durante la sostituzione, si possono rinominare le variabili legate (es. $\lambda x.x \equiv \lambda y.y$).

  - è buona pratica seguire la "convenzione di Barendregt", che stabilisce che, all'interno di un termine lambda o di un insieme di termini, tutte le variabili legate devono avere nomi distinti tra loro e diversi da tutte le variabili libere presenti nel termine stesso

- I termini tali che ($FV(M) = \emptyset$) (λ-termini chiusi), ovvero senza variabili libere, sono chiamati **combinatori**

2.2.1 Combinatori noti e strutture di controllo

Il λ-calcolo puro non ha tipi di dato predefiniti: tutto (booleani, numeri, if-then-else) deve essere codificato tramite funzioni.

- **Identità (I):** $I \equiv \lambda x.x$
- **Cancellatori (booleani):**

  True (K): $K \equiv \lambda xy.x$ (prende due argomenti e restituisce il primo)

  False (O): $O \equiv \lambda xy.y$ (prende due argomenti e restituisce il secondo)
- **If-Then-Else:** grazie alla definizione dei booleani, la struttura condizionale è codificata come un'applicazione: $if\ x\ then\ M\ else\ N \equiv xMN$. Se x è True (K) selezionerà M, se è False (O) selezionerà N.
- **Compositori (S):** $S \equiv \lambda xyz.xz(yz)$ (si può dimostrare che $SKK \approx I$).
- **Duplicatori e divergenza:**

  $\omega \equiv \lambda x.xx$ (auto-applicazione)

  $\Omega \equiv \omega\omega \to \omega\omega \to \dots$ rappresenta la **funzione indefinita**, il loop infinito (non tipabile)

2.2.1.1 Numeri di Church e iteratori

I **Numerali di Church** rappresentano i numeri naturali come funzioni di ordine superiore che *iterano* un'operazione. Il numero n è una funzione che prende una funzione (es. successore) s e un valore di partenza (es. zero) z, e applica s ad z esattamente n volte

$$\underline{n} \equiv \lambda sz.s(s(\dots(sz)\dots)) \quad [n \text{ applicazioni}]$$

- $\underline{0} \equiv \lambda sz.z$ (equivale a `False`)
- $\underline{3} \equiv \lambda sz.s(s(s(z)))$

Per calcolare il successore ($+1$) di un numero di Church, basta creare una funzione che applica s una volta in più: $succ \equiv \lambda xsz.s(xsz)$ (o anche $\lambda xsz.xs(sz)$)

I numerali di Church non sono quindi solo dati, ma **iteratori intrinseci**.

2.3 Ricorsione e Combinatori di Punto Fisso

Per definire funzioni ricorsive complesse il λ-calcolo necessita di un principio di ricorsione generale.

> Thm. 1: **Combinatore di Punto Fisso**
>
> Nel λ-calcolo esiste un termine Y (detto **Combinatore di Punto Fisso**) tale che, per ogni termine M, si ha $YM \to M(YM)$.

Il combinatore permette di trovare quindi il punto fisso di una funzione (quel valore x tale che $f(x) = x$)

Due combinatori di punto fisso sono:

- **Combinatore di Turing (Y_T):** definito come $Y_T = \theta\theta$ dove $\theta \equiv \lambda xy.y(xxy)$.

$$\begin{aligned} Y_T M &\equiv (\theta\theta)M \\ &\equiv ((\lambda xy.y(xxy))\theta)M \\ &\to_\beta (\lambda y.y(\theta\theta y))M \\ &\to_\beta M(\theta\theta M) \\ &\equiv M(Y_T M) \end{aligned}$$

- **Combinatore di Curry (Y_C):** definito come $Y_C \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$. Se applicato all'identità I, questo combinatore riduce a Ω (divergenza).

$$\begin{aligned} Y_C M &= (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))M \\ &\to_\beta (\lambda x.M(xx))(\lambda x.M(xx)) \\ &\to_\beta M((\lambda x.M(xx))(\lambda x.M(xx))) \\ &\equiv M(Y_C M) \end{aligned}$$

(l'ultima equivalenza $\equiv$ vale perché $(\lambda x.M(xx))(\lambda x.M(xx))$ è la forma ridotta di $Y_C M$)

Thm. 2: **Tesi di Church-Turing**

Ogni funzione calcolabile è λ-definibile.

3

Temi avanzati di programmazione funzionale

3.1 Fusion law per foldr

La **Fusion Law** per `foldr` è un principio di induzione generalizzato che permette di trasformare la composizione di una funzione con una `foldr` in un'unica fold. Essa stabilisce le condizioni per cui vale l'uguaglianza:

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

L'obiettivo è quindi trovare funzioni o valori `h` e `b` tali che l'applicazione di `f` al risultato di una `foldr` sia equivalente ad una nuova `foldr` che lavora direttamente su di essi - l'utilità computazionale risiede nella capacità di "fondere" due passaggi (una trasformazione `f` applicata dopo un fold con `g`) in un unico ciclo sulla struttura dati, evitando allocazioni intermedie.

Thm. 3: **Fusion law**

Siano `f`, `g`, `h` funzioni e `a`, `b` valori. L'uguaglianza `f . foldr g a = foldr h b` è garantita se sono soddisfatti i seguenti tre requisiti:

(1) `f` è stretta (`f undefined = undefined`) (così che valga anche nel caso di liste parziali/indefinite)

(2) **caso base**: `f a = b` - `f` applicata all'accumulatore iniziale della prima `foldr` (`a`) deve dare come risultato l'accumulatore iniziale della seconda (`b`)

(3) **passo induttivo** (condizione di fusione): per ogni `x`, `y`, deve valere:
`f (g x y) = h x (f y)` - in questo modo, l'operazione `h` del nuovo `foldr` riproduce esattamente l'effetto di `g` seguito da `f`

parallelo con un omorfismo

La fusion law può essere interpretata come la dimostrazione che `f` agisce come un **omomorfismo** tra due strutture algebriche di calcolo:

- **preservazione dell'identità**: `f a = b` mappa l'elemento neutro (accumulatore iniziale) del primo dominio in quello del secondo, analogamente a $f(1_A) = 1_B$.
- **preservazione dell'operazione**: La condizione `f(g x y) = h x (f y)` ricalca la proprietà omomorfica $f(a \bullet b) = f(a) \circ f(b)$.

  - *sorgente* (`g x y`): rappresenta l'operazione di combinazione di un elemento `x` con il risultato parziale `y` tramite la funzione `g`
  - *destinazione* (`h x (f y)`): rappresenta la nuova operazione `h` che combina `x` con il risultato già trasformato da `f`
- **NB**: in `h x (f y)`, l'elemento `x` non viene trasformato da `f` - nella struttura della lista, `x` è un dato "guida" di tipo A, mentre `y` è il risultato della ricorsione (accumulatore) di tipo B. La fusion law mira a trasformare il *risultato complessivo* del fold, non i singoli elementi contenuti nella lista.

Derivazione formale

Per dimostrare la validità dell'uguaglianza `f . foldr g a = foldr h b`, analizziamo separatamente i due lati della relazione applicando i principi di induzione sulla struttura della lista:

Ricordando la definizione di `foldr` basata sulla decomposizione della lista:

- `foldr g a [] = a`
- `foldr g a (x:xs) = g x (foldr g a xs)`

Caso `[]` (base):

- **sinistra**: `(f . foldr g a) []`
  `= f (foldr g a [])` *(def di .)*
  `= f a` *(def di foldr)*
- **destra**: `foldr h b []`
  `= b` *(def di foldr)*
- da cui deriva la prima condizione: `f a = b`

Caso `(x:xs)` (induttivo):

- **sinistra**: `(f . foldr g a) (x:xs)`
  `= f (foldr g a (x:xs))` *(def di .)*
  `= f (g x (foldr g a xs))` *(def di foldr)*
- **destra**: `foldr h b (x:xs)`
  `= h x (foldr h b xs)` *(def di foldr)*
  `= h x (f (foldr g a xs))` *(induzione)*
- ponendo `y = foldr g a xs`, deriviamo la condizione di fusione:
  `f (g x y) = h x (f y)`

3.2 TODO: call-by-name + laziness (08)

3.3 TODO: funzioni strette e non strette

3.4 TODO: tecniche di ottimizzazione (09)

3.5 TODO: scanl e scanr (09bis)

3.6 Strutture dati infinite

La valutazione lazy permette di maneggiare in modo astratto ed elegante **strutture dati infinite**.

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

> Def. 1: **Massima vs minima algebra chiusa**
>
> - *Minima algebra chiusa* (induzione): insieme di tutti i dati finiti che si possono costruire partendo dai casi base (esempio: numeri naturali di Peano)
> - *Massima algebra chiusa* (co-induzione): include anche gli oggetti costruiti applicando i costruttori infinite volte - i tipi di Haskell ammettono questa natura infinita
>
>   per esempio, la co-algebra dei naturali includerebbe anche il 'naturale infinito'
>
> $$\omega = S(S(\dots)) = S^{\infty}(\dots)$$

Nel caso delle liste, le liste *finite* sono la minima algebra chiusa rispetto a `:` e `[]`, mentre le liste infinite (o **stream**) sono la relativa massima algebra.

Eliminando i costruttori costanti (casi base) dalle algebre induttive, si possono "isolare" gli elementi infiniti:

- eliminando `zero` dai naturali, si ottiene l'algebra che contiene solo il naturale infinito
- eliminando `[]` dalle liste, si ottiene l'algebra che contiene solo stream

3.6.1 Definizioni naif e circolari

Il modo che si sceglie per generare stream ne cambia drasticamente l'efficienza.

Possiamo definire le implementazioni di stream in due categorie: naif (basate sul ricalcolo), e circolari (basate sulla condivisione)

3.6.1.1 Definizioni naif

Una definizione naif richiama esplicitamente se stessa passando nuovamente gli argomenti

esempio: potenze e iterate

```
1 powers n = 1 : map (n*) (powers n)
2
3 myIterate f x = x : map f (myIterate f x)
```

Queste implementazioni hanno complessità quadratica ($O(n^2)$), in quanto, per calcolare l'n-esimo elemento, il programma deve ripercorrere tutti i passaggi precedenti.

La complessità è causata da un'assenza di memoizzazione del lavoro già fatto: ogni volta che si chiede un nuovo elemento viene generata una nuova chiamata alla funzione, creando una catena infinita di funzioni annidate (`map (*2) (map (*2) (map (*2) ...))`).

3.6.1.2 Definizioni circolari e lineari

Una definizione si dice circolare quando la struttura dati viene definita in termini di se stessa, creando un puntatore in memoria (Graph Reduction) invece di generare nuove chiamate a funzione. Questo garantisce una complessità lineare ($O(n)$) perché ogni elemento viene calcolato una volta sola.

Esistono due modi principali per implementarle:

- **livello globale**: senza parametri, il nome della lista è sufficiente per creare il riferimento circolare

```haskell
ones = 1 : ones
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

- **all'interno di funzioni**: si usano `where` o `let` per "fissare" i parametri e definire la lista ricorsivamente rispetto alla variabile locale

```haskell
powers' n = ps where ps = 1 : map (*n) ps
myIterate' f x = xs where xs = x : map f xs
```

È utile saper distinguere tra velocità (linearità) e struttura di memoria (circolarità). Non tutte le ricorsioni dirette sono inefficienti.

```haskell
iterate1 f x = x : iterate1 f (f x)
```

Questa definizione è lineare ($O(n)$) pur non essendo circolare. Ogni passo produce un elemento applicando la funzione all'argomento corrente, senza accumulare operazioni sospese (come catene di `map`) che invece renderebbero la computazione quadratica.

> Produttività
>
> La "regola d'oro" per gli stream è garantire la **produttività**: il processo di generazione deve consumare meno input di quanto output produce.
>
> - esempio non produttivo: `incstnts = (head incstnts) : (tail incstnts)`.
>
>   la definizione richiede di consumare la testa della lista prima ancora di averla generata, portando a una mancata terminazione (il processo "si mangia la coda")

3.7 Liste infinite come limiti

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

Per gestire l'infinito e il calcolo parziale, Haskell usa il simbolo $\perp$ (bottom, anche usato per la contraddizione in logica), che rappresenta una computazione che non termina o che ha dato errore (corrisponde ad `undefined`).

L'esecuzione di un programma può essere vista come un **accumulo di informazioni**:

- un valore non ancora calcolato ($\bot$) non ha informazioni
- man mano che il programma avanza, $\bot$ viene sostituito da valori reali (es: $1 : \bot$, poi $1 : 2 : \bot$)

Possiamo quindi definire una relazione di ordine parziale $\sqsubseteq$, l'**ordine di approssimazione**.

Diciamo che $x \sqsubseteq y$ (x approssima y) se y contiene almeno tutte le informazioni di x.

Per esempio, una lista infinita `[1..]` può essere vista come il limite di una sequenza di approssimazioni che diventano sempre più precise:

$$\bot \sqsubseteq 1 : \bot \sqsubseteq 1 : 2 : \bot \sqsubseteq 1 : 2 : 3 : \bot \ldots$$

Il limite di questa catena è la lista infinita completa.

(Invece, una lista come $\bot, 1 : \bot, 2 : 1 : \bot, 3 : 2 : 1 : \bot, \ldots$ non converge a nessun limite: non si stabilizza a nessun valore e offre informazioni contraddittorie).

3.7.1 Domini, ordine e limiti

Vediamo nel dettaglio come Haskell misura l'informazione prodotta da un programma.

3.7.1.1 Tipi semplici

Per quanto riguarda i tipi semplici, l'informazione è un "tutto o niente": o il valore è calcolato, o non lo è.

La relazione $\sqsubseteq$ per i tipi semplici è *piatta* (**flat domain**):

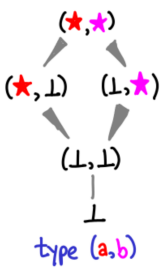
$$x \sqsubseteq y \text{ sse } x = \bot \text{ oppure } x = y$$

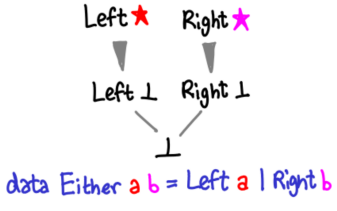
Si ha quindi per esempio $\bot \sqsubseteq$ `True`, ma non ci sono relazioni tra `True` e `False`.

3.7.1.2 Costruttori di tipo

L'ordine si estende naturalmente ai tipi composti. L'idea è che una struttura è "più definita" di un'altra se i suoi componenti lo sono.

- *Coppie*: $(x, y) \sqsubseteq (x', y') \iff x \sqsubseteq x' \wedge y \sqsubseteq y'$
- *Either*:
  - $\bot \sqsubseteq$ `Left` $\bot$, `Right` $\bot$
  - `Left` $x \sqsubseteq$ `Left` $x' \iff x \sqsubseteq x'$
  - `Right` $x \sqsubseteq$ `Right` $x' \iff x \sqsubseteq x'$



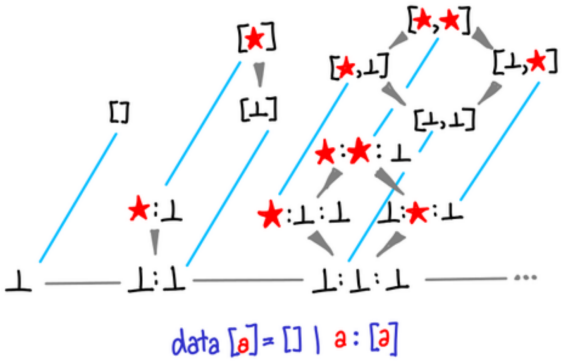


3.7.1.3 Liste e funzioni

Per le liste si ha:

- $\bot \sqsubseteq xs$
- $[\,] \sqsubseteq xs \iff xs = [\,] \wedge (x : xs) \not\sqsubseteq [\,]$
- $x : xs \sqsubseteq y : ys \iff x \sqsubseteq y \wedge xs \sqsubseteq ys$

Per esempio, $[1, \bot, 3]$ e $1 : 2 : \bot$ sono approssimazioni di $[1, 2, 3]$.



Invece, una funzione f ne approssima un'altra g se per ogni possibile input x, l'output di f approssima quello di g:

$$\text{date } f, g : a \to b \text{ si ha } f \sqsubseteq_{a\to b} g \iff \forall x. fx \sqsubseteq_b gx$$

3.7.1.4 Ordine e limiti

> Def. 2: **Complete Partial Order**
>
> Se abbiamo una catena di approssimazioni $x_1 \sqsubseteq x_2 \sqsubseteq x_3 \ldots$, questa tenderà ad un limite $\sqcup x = \lim_{n\to\infty} x_n$ che soddisfa le seguenti 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 il sup)
>
> (simile al lemma di Zorn ma più forte)
>
> Un ordinamento che soddisfa queste proprietà si definisce **Complete Partial Order**

I tipi per domini di funzioni computabili (e quindi eseguibili da una macchina) sono solitamente CPO.

> Def. 3: **Funzioni computabili**
>
> Una funzione computabile gode di due proprietà:
>
> (1) **monotonia**: $x \sqsubseteq y \implies fx \sqsubseteq fy$
>
> (2) **continuità**: $f(\sqcup x) = \sqcup fx$

> Approfondimento: computabilità logica vs. semantica
>
> È interessante comparare la definizione di funzione computabile fornita dalla teoria dei domini

a quella classica della logica matematica (funzioni μ-ricorsive).

Mentre la definizione logica è di tipo costruttivo - identifica la classe delle funzioni calcolabili come la minima classe ottenuta partendo da funzioni base (zero, successore, proiezioni) e applicando operatori di chiusura (composizione, minimalizzazione) -, la definizione basata sui CPO è di tipo semantico. Essa stabilisce i vincoli necessari affinché una funzione possa operare su informazioni parziali o infinite:

- La *monotonia* garantisce che la computazione sia un processo ad accumulo di informazione: se l'input diventa più definito ($x \sqsubseteq y$), l'output non può diminuire o cambiare l'informazione già prodotta ($fx \sqsubseteq fy$)
- La *continuità* formalizza la natura finitaria della calcolabilità: il valore di una funzione su un input infinito (limite) deve essere determinato esclusivamente da ciò che la funzione produce sulle approssimazioni finite di quell'input ($f(\sqcup x) = \sqcup fx$)

In questo contesto, la parzialità delle funzioni logiche viene "internalizzata" nel dominio tramite l'elemento $\bot$ (bottom), che rappresenta il livello minimo di informazione o una computazione non terminata. Questo permette di trattare la non-terminazione non come un'assenza di valore, ma come un valore matematico ben definito su cui è possibile applicare i teoremi di punto fisso per dare un senso alla ricorsione.

Alcuni esempi di funzioni non computabili sono:

- $f(x) = \begin{cases} 0 & x = \bot \\ 1 & x \neq \bot \end{cases}$
  - non è computabile perché non è monotona: $\bot \sqsubseteq x$, ma $0 \not\sqsubseteq 1$
- $f(xs) = \begin{cases} \bot & xs \text{ è finita o parziale} \\ 1 & xs \text{ è infinita} \end{cases}$
  - non è computabile perché non è continua: $\sqcup g\ x_n = \bot \neq 1 = g(\sqcup\ x_n)$

3.8 Teoremi di punto fisso

Nella semantica denotazionale, l'esecuzione di un programma ricorsivo è vista come la ricerca di un "punto fisso" di una funzione. Per formalizzare questo concetto, dobbiamo estendere la nozione di CPO a strutture algebriche più ricche.

Def. 4: **Reticolo Completo**

Un **reticolo completo** (Complete Lattice) $(L, \sqsubseteq)$ è un insieme parzialmente ordinato in cui *ogni* sottoinsieme $A \subseteq L$ possiede sia un estremo inferiore che un estremo superiore:

- Estremo superiore (Supremum): $\sqcup A = \min\{x \mid \forall a \in A.\ x \geq a\}$
- Estremo inferiore (Infimum): $\sqcap A = \max\{x \mid \forall a \in A.\ x \leq a\}$

Avere un reticolo completo ci permette di enunciare due teoremi fondamentali che garantiscono l'esistenza (e la calcolabilità) dei punti fissi per le funzioni su questi domini (un valore x è un *punto fisso* per una funzione f se $f(x) = x$).

Def. 5: **Teorema di Knaster-Tarski**

In un reticolo completo $(L, \sqsubseteq)$, **ogni funzione monotona** ammette un massimo e un minimo punto fisso.

Il teorema di Knaster-Tarski assicura l'esistenza del punto fisso, ma non dice come trovarlo. Il teorema di Kleene ci fornisce invece un metodo costruttivo.

Def. 6: **Teorema di Kleene (del minimo punto fisso)**

Se una funzione f è **continua**, il suo minimo punto fisso (Least Fixed Point, LFP) può essere costruito iterativamente partendo dal livello minimo di informazione ($\bot$).

Esso è definito come il limite delle applicazioni ripetute di f:

$$\text{LFP}(f) = f^{\omega}(\bot) = \lim_{n \to \infty} f^n(\bot) = \sqcup\{\bot, f(\bot), f(f(\bot)), \dots\}$$

proof

Sappiamo che la sequenza $\bot \sqsubseteq f(\bot) \sqsubseteq f(f(\bot)) \dots$ è una catena monotona crescente e che quindi (poiché ci troviamo in un CPO) ha un estremo superiore.

Possiamo dimostrare che questo estremo superiore f^{ω} è un punto fisso sfruttando la continuità:

$$\begin{aligned}
f^{\omega} &= \sup\{f^i(\bot) \mid i < \omega\} && \text{(per definizione di } f^{\omega}) \\
&= \sup\{f(f^i(\bot)) \mid i < \omega\} && \text{(spostando l'indice di 1 passo)} \\
&= f(\sup\{f^i(\bot) \mid i < \omega\}) && \textbf{(per la continuità di } f) \\
&= f(f^{\omega}) && \text{(sostituendo la definizione di } f^{\omega})
\end{aligned}$$

Poiché $f(f^{\omega}) = f^{\omega}$, f^{ω} è per definizione un punto fisso di f.

Significato computazionale dei teoremi di punto fisso

Quando scriviamo una definizione ricorsiva come `ones = 1 : ones`, stiamo implicitamente definendo un'equazione: $x = f(x)$, dove la funzione è $f(x) = 1 : x$ - stiamo quindi cercando il punto fisso di f.

Il Teorema di Kleene ci spiega esattamente cosa fa la valutazione lazy: il computer costruisce la lista infinita partendo dal non-calcolato ($\bot$) e applicando la funzione in modo continuo:

(1) $f^0(\bot) = \bot$ *(nessuna informazione)*

(2) $f^1(\bot) = 1 : \bot$ *(primo elemento noto)*

(3) $f^2(\bot) = 1 : 1 : \bot$ *(secondo elemento noto)*

Il limite di questo processo all'infinito (f^{ω}) è proprio la nostra lista infinita. La continuità garantisce che per estrarre una quantità di informazione finita dall'output (es. `take 2 ones`), il computer debba compiere solo un numero finito di passi, senza bloccarsi cercando di valutare l'intera struttura infinita.

3.9 TODO: induzione e coinduzione

3.10 Funtori e Applicativi

> Concetti principali
>
> Sia i Funtori che gli Applicativi servono a gestire computazioni all'interno di un **contesto**, ma con "poteri" diversi:
>
> - **Functors**: permettono di applicare una funzione "pura" a un valore nel contesto - modificano il contenuto ma non la struttura del contesto;
> - **Applicatives**: permettono di applicare funzioni che sono esse stesse dentro un contesto a valori nel contesto - sono necessari per gestire funzioni con più argomenti;

classe	operatore	scopo
`Functor`	`fmap :: (a -> b) -> f a -> f b`	trasformazione (1 argomento)
`Applicative`	`<*> :: f (a -> b) -> f a -> f b`	combinazione (n argomenti)

3.10.1 Funtori

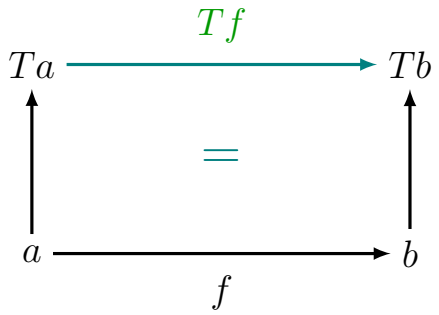
L'idea alla base di un **funtore** è la generalizzazione del funzionale `map`, tipicamente utilizzato per le liste, a tutti i possibili costruttori di tipo.

Se si ha:

- una funzione $f :: a \to b$
- un costruttore di tipo T che trasforma un tipo a in un tipo Ta (come per esempio `[a]` oppure `Maybe a`)

Si può ottenere in modo canonico una funzione

$$Tf :: Ta \to Tb$$



> Nel mondo categoriale, il concetto di "funtore" descrive un passaggio tra due categorie: dati due oggetti (tipi come a o b), un funtore agisce come un costruttore di tipo (T) - prende un oggetto a nella categoria C e lo trasforma in un oggetto Ta nella categoria D.
>
> - il punto importante è che un funtore non mappa solo i tipi, ma anche le relazioni tra essi: se esiste una funzione $f : a \to b$, il funtore deve fornire un modo per ottenere una funzione corrispondente Tf che operi sui tipi $Ta \to Tb$

- un funtore è quindi una **mappa che preserva la struttura**

La classe `Functor` richiede che sia definita una funzione `fmap`, che rende commutativo il passaggio dai tipi base ai tipi "inscatolati"

```
class Functor t where
    fmap :: (a -> b) -> t a -> t b
```

- il tipo di `fmap` è parametrico, oltre che nei tipi `a` e `b`, anche rispetto a un **costruttore di tipo** `t` (si vede che `t` deve essere un costruttore di tipo perché si applica ad altri tipi)

Esempi

Alcuni esempi noti di istanze di funtori sono:

- le liste

```
class Functor [] where
    fmap = map
```

- il tipo `Maybe`

```
class Functor Maybe where
    fmap f Nothing = Nothing
    fmap f (Just x) = Just (f x)
```

- gli alberi binari

```
class Functor BinTree where
    fmap f (F x) = F (f x)
    fmap f (R r lft rgt) =
        R (f r) (fmap f lft) (fmap f rgt)
```

Definire un tipo come istanza di `Functor` permette di scrivere codice generico che funziona su diverse strutture dato.

Per esempio, si può definire una funzione `inc` che incrementa i valori all'interno di qualsiasi funtore:

```
inc :: Functor t => t Int -> t Int
inc = fmap (+1)
```

3.10.1.1 Functor laws

Affinché un funtore sia definito correttamente, è necessario che rispetti le due functor laws:

(1) `fmap id = id`

(2) `fmap (f . g) = fmap f . fmap g`

Nota bene: queste leggi non possono essere verificate dal type-checker, ma sono responsabilità del programmatore Haskell !

> Funtori come omomorfismi
>
> Possiamo notare, osservando le *functor laws*, che un funtore è effettivamente un **omomorfismo tra categorie** (un omomorfismo "classico" preserva l'operazione di gruppo, mentre un funtore preserva la "struttura" di una categoria, ovvero gli oggetti - i tipi -, e i morfismi - le funzioni.)

3.10.2 Applicativi

Un normale funtore permette di usare `fmap` con un solo argomento, ma è naturale voler generalizzare: si vuole considerare una forma astratta di applicazione, che generalizzi l'usuale applicazione di funzioni. Per esempio, si vuole poter propagare un fallimento nel caso di `Maybe`, o applicare una funzione n-aria a valori "inscatolati" senza dover istanziare n funtori.

Haskell definisce quindi la classe `Applicative`.

```haskell
class Functor t => Applicative t where
    pure :: a -> t a
    (<*>) :: t (a -> b) -> t a -> t b
```

Un applicativo estende un funtore, e richiede due operatori fondamentali:

- `pure`: prende un valore "normale" e lo immerge nel contesto (lo "impacchetta")

  per esempio, per `Maybe`, `pure x` è `Just x`
- `<*>` (*splat* o *apply*): prende una funzione "impacchettata" in un contesto (`t (a -> b)`) e un valore anch'esso in un contesto (`t a`), "scarta" concettualmente entrambi i contesti, applica la funzione al valore (gestendo gli effetti del contesto), produce il risultato e lo "incarta" nel contesto (`t b`)

> Esempio: Maybe
>
> Possiamo istanziare `Maybe` anche come applicativo:
>
> ```haskell
> instance Applicative Maybe where
>     pure = Just
>     Nothing <*> _ = Nothing -- se uno dei due è Nothing, entrambi sono Nothing
>     (Just g) <*> mx = fmap g mx
>     -- <*> prende una funzione e un valore in un contesto, quindi
>     -- per es. Just (+3) <*> Just 2 = Just 5
> ```

3.10.2.1 Liste come applicativi

L'istanza del Prelude delle liste non si comporta come una `zipWith` nell'applicare le funzioni presenti in una lista, ma modella un non-determinismo. Una lista non viene vista come una sequenza, ma come un valore che può assumere più risultati posssibili contemporaneamente. Quindi, quando si applica una lista di funzioni ad una lista di valori, l'applicativo contiene tutte le combinazioni possibili (prodotto cartesiano).

```haskell
instance Applicative [] where
    -- pure : a -> [a]
    pure x = [x]

    -- <*> : [a -> b] -> [a] -> [b]
    gs <*> xs = [g x | g <- gs, x <- xs]
    -- ^ prodotto cartesiano
```

Se eseguiamo quindi `pure (*) <*> [1,2] <*> [3,4]`:

- `pure (*)` crea `[(*)]`
- `[(*)] <*> [1,2]` produce una lista di funzioni `[(1*), (2*)]`
- `[(1*), (2*)] <*> [3,4]` applica entrambe le funzioni ad entrambi i numeri

  si ottiene quindi `1*3, 1*4, 2*3, 2*4` → `[3, 4, 5, 6]`

A volte si vuole però che le liste si comportino come `zipWith` - per questo, haskell introduce `ZipList`

```haskell
 -- per poter definire un'altra istanza di Applicative senza conflitti
newtype ZipList a = Z [a]

instance Applicative ZipList where
    -- pure : a -> ZipList a
    pure x = Z (repeat x) -- crea una lista infinita di x

    -- <*> : [a -> b] -> [a] -> [b]
    Z fs <*> Z xs = Z (zipWith (\f x -> f x) fs xs)
```

- `pure` è `repeat` così che si adatti a qualsiasi lunghezza della lista di valori con cui verrà accoppiata (se restituisse un solo elemento, `zipWith` si fermerebbe subito)
- `<*>` è `zipWith` - applica la funzione in posizione 1 al valore in posizione 1 ecc

3.10.2.2 Applicative laws

Come i funtori, anche gli applicatori hanno una serie di regole da rispettare:

(1) `<*>` **preserva l'identità**

`pure id <*> x = x`

se prendiamo la funzione identità, la inseriamo in un contesto e la applichiamo ad un valore `x` già contestualizzato, il risultato deve rimanere `x`

(2) `<*>` **preserva l'applicazione di funzioni**

`pure (g x) = pure g <*> pure x`

applicare `g` ad un valore `x` e poi "impacchettare" il risultato dà lo stesso esito che applicare la versione "impacchettata" di `g` alla versione "impacchettata" di `x`

(3) **non conta l'ordine** con cui si fa embedding nel contesto

`x <*> pure y = pure (\g -> g y) <*> x`

(4) **composizione** di `<*>`

`x <*> (y <*> z) = (pure (.) <*> x <*> y) <*> z`

(forma di associatività per `<*>`)

la composizione di funzioni `(.)` può essere "lifted" a livello applicativo per garantire che la sequenza delle applicazioni sia coerente

- ovvero, applicare due funzioni una dopo l'altra è identico a comporle prima in un'unica funzione e poi applicarla:

  `x <*> (y <*> z)`

  prima si applica la funzione `y` al valore `z` (ottenendo un risultato nel contesto), e poi si applica `x` a tale risultato - è quindi `x(y(z))`

  `(pure (.) <*> x <*> y) <*> z`

  qui l'operatore di composizione standard `(.)` viene "sollevato" (lifted) nel contesto, permettendo di fondere le due funzioni `x` e `y` dentro il contesto - si crea quindi una funzione composta che sarà applicata a `z`

  quindi, si applica `x . y` a `z`, ottenendo anche qui `x(y(z))`

3.11 TODO: monadi (14)

3.12 TODO: monadi I/O ecc (15)