Tecniche di Programmazione Funzionale e Imperativa

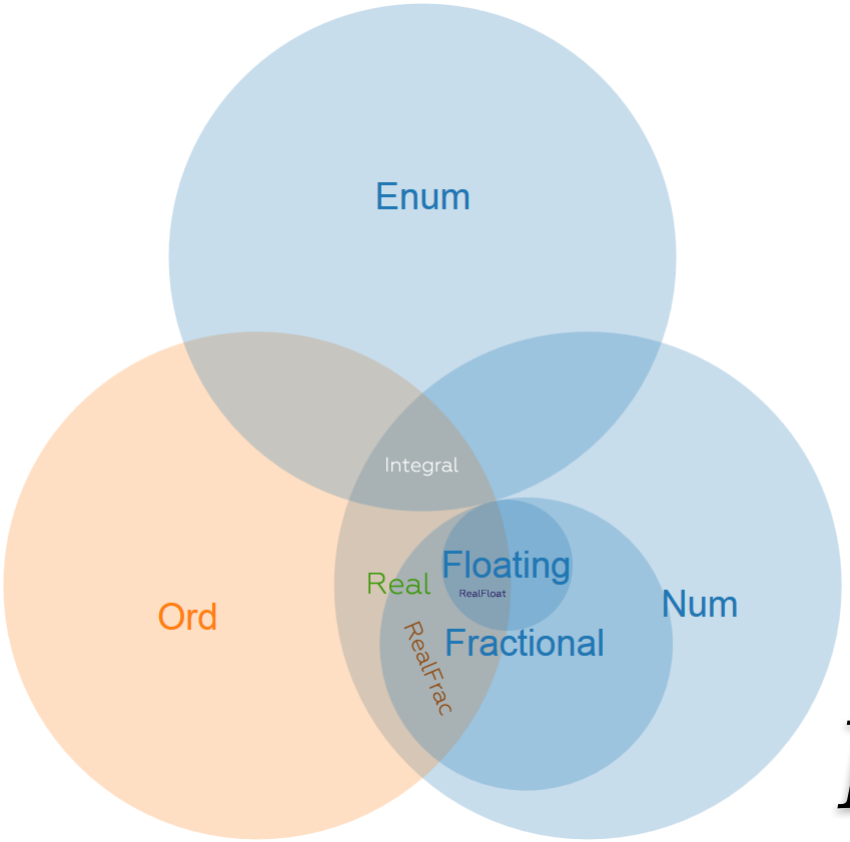
Ivano Salvo

Tipi Strani, ma di Classe

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

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **5**, 8 marzo 2022



Lezione 5a

Classi e istanze in Haskell

Classi e subtyping (1)

Abbiamo solo accennato alle **classi**, cioè alle annotazioni di tipo prima della freccia **=>**. Scriviamo la funzione **mcd**.

Interroghiamo l'interprete sul **tipo** che viene **inferito** da Haskell: potremmo aspettarci semplicemente **Integer -> Integer -> Integer**, ma il tipo calcolato è decisamente più generale.

La funzione mcd può funzionare su tutti i tipi che sono allo stesso tempo sottotipo di **Ord** (perché testo il < e ==) e di **Num** (perché faccio l'operazione aritmetica -).

Questo è il significato di quanto appare prima di **=>**: analogo alle **interfacce** di Java.

```
 -- mcd di Euclide, versione additiva
mcd x y
 | x == y    = x
 | x  < y    = mcd (y-x) x
 | otherwise = mcd (x-y) y

>:t mcd
mcd :: (Ord a, Num a) => a -> a -> a
```

Classi e subtyping (2)

Ci sono esempi ancora più banali. Proviamo a chiedere a **GHCi** il tipo del numero 3.

La costante `3` ha un **tipo polimorfo**, **ma non completamente polimorfo**, perché potrebbe essere un `Integer`, un `Float`, etc.

Similmente, per `(+)` e altre operazioni aritmetiche definite **su tutti i tipi numerici**.

Leggermente diversa la questione per la divisione `(/)` e per la divisione intera, `div`. Trattasi di **polimorfismo limitato**.

```
 -- tipi numerici `strani`
> :t 3
3 :: Num a => a

> :t (+)
(+) :: Num a => a -> a -> a

>:t (/)
(/) :: Fractional a => a -> a -> a

>:t div
div :: Integral a => a -> a -> a
```

Classi predefinite: Eq

Le **classi** possono essere viste in Haskell come una **collezione** (oppure una **classificazione**) **di tipi**, un po' come i **tipi** sono una collezione (o classificazione) di **valori**.

Fin d'ora, è bene ricordare che questa classificazione **non è** semplicemente **insiemistica**, ma essenzialmente **algebrica**.

Cominciamo **Equality Types**, **`Eq`**: comprende tutti i tipi su cui è definita la relazione di **uguaglianza `(==)`** e la sua negazione **`(/=)`**. Notare che è parametrica rispetto al tipo **`a`**.

Tutti i tipi base (`Bool`, `Char`, `String`, `Int`, `Integer`, `Float` e `Double`) sono istanze di `Eq`. **Liste** e **tuple** sono anch'esse **equality types** a patto che siano costruite a partire da **equality types**.

`(/=)` è definito **una volta per tutte**, come negazione di **`(/=)`**.

```
 -- definizione della classe Eq
class Eq a where
   (==), (/=) :: a -> a -> Bool
-- le classi possono contenere `codice`
   x /= y = not (x == y)
```

Classi predefinite: Ord

Le **classi** possono formare una **gerarchia**. Ad esempio la classe **`Ord`** è una **sottoclasse** di **`Eq`**, in quanto gli insiemi (parzialmente) ordinati con un ≤ hanno anche una nozione di ugualglianza.

Al solito, "estendere" significa avere più operazioni/predicati definiti. In questo caso **`<=`**, **`<`**, **`>=`**, **`>`**, **`min`** e **`max`**.

Tutti i **tipi base sono istanze** di **`Ord`**. **Liste** e **tuple** sono anch'esse istanze a patto che siano costruite a partire da **ordered types**: viene esteso con l'**ordine lessicografico**.

```
-- definizione di Ord come estensione di Eq
class Eq a => Ord a where
   (<),(<=),(>=),(>)   :: a -> a -> Bool
   min, max            :: a -> a -> a

   min x y | x <= y = x | otherwise y
   max x y | x <= y = y | otherwise x
   x <= y = x < y || x==y
   x >= y = x > y || x==y
   x > y  = y >= x && y /= x
```

È sufficiente fornire il codice per <, il resto è interdefinito

```
-- ordine lessicografico su liste:
> [1,2] < [1,2,3]
True
> [0..] < [1]
True
```

Classi predefinite: Num e Show

Ci sono **numerose classi** numeriche in Haskell, e al top della **gerarchia** c'è la più generica **Num**. Un tipo è istanza di **Num** se implementa le usuali operazioni aritmetiche, valore assoluto, segno e opposto.

Show è la classe che implementano tutti i tipi che si possono stampare. Contiene essenzialmente un metodo, **show**.

La classe **Read** permette di trasformare una stringa in un valore.

```
 -- definizione di Num
class (Eq a, Show a) => Num a where
   (+), (-), (*)       :: a -> a -> a
   negate, abs, signum :: a -> a
   fromInteger         :: Integer -> a
```

```
 -- definizione di Show
class Show a where
   show :: a -> String
```

```
 -- definizione di Read
class Read a where
   read :: String -> a
```

Esempi di Show

Possiamo capire il messaggio di **warning** che viene visualizzato se provate a **valutare una funzione**.

Quando valutate un'espressione sull'interprete, viene calcolato il **valore**, chiamato il metodo **`print`** che a sua volta chiama **`show`** per trasformare il valore in una stringa da mandare a video.

Ma **sulle funzioni** (specie se generiche) **non è definito** nessun metodo **`show`**.

⏵ **Esercizio**: elaborate una strategia per stampare i numerali di Church, che sono funzionali, ma sono anche numeri.

```
>\x -> x
<interactive>:7:1:
    No instance for (Show (t0 -> t0)) arising from a use of ‘print’
    In a stmt of an interactive GHCi command: print it
```

Altre classi numeriche

Altre interessanti classi numeriche sono:

Tipi **Interi**, **`Integral`** che definisce tipi numerici che hanno inoltre le seguenti funzioni (esempio **`Int`**, **`Integer`**, **`Word`**):

```
div, mod :: a -> a -> a
```

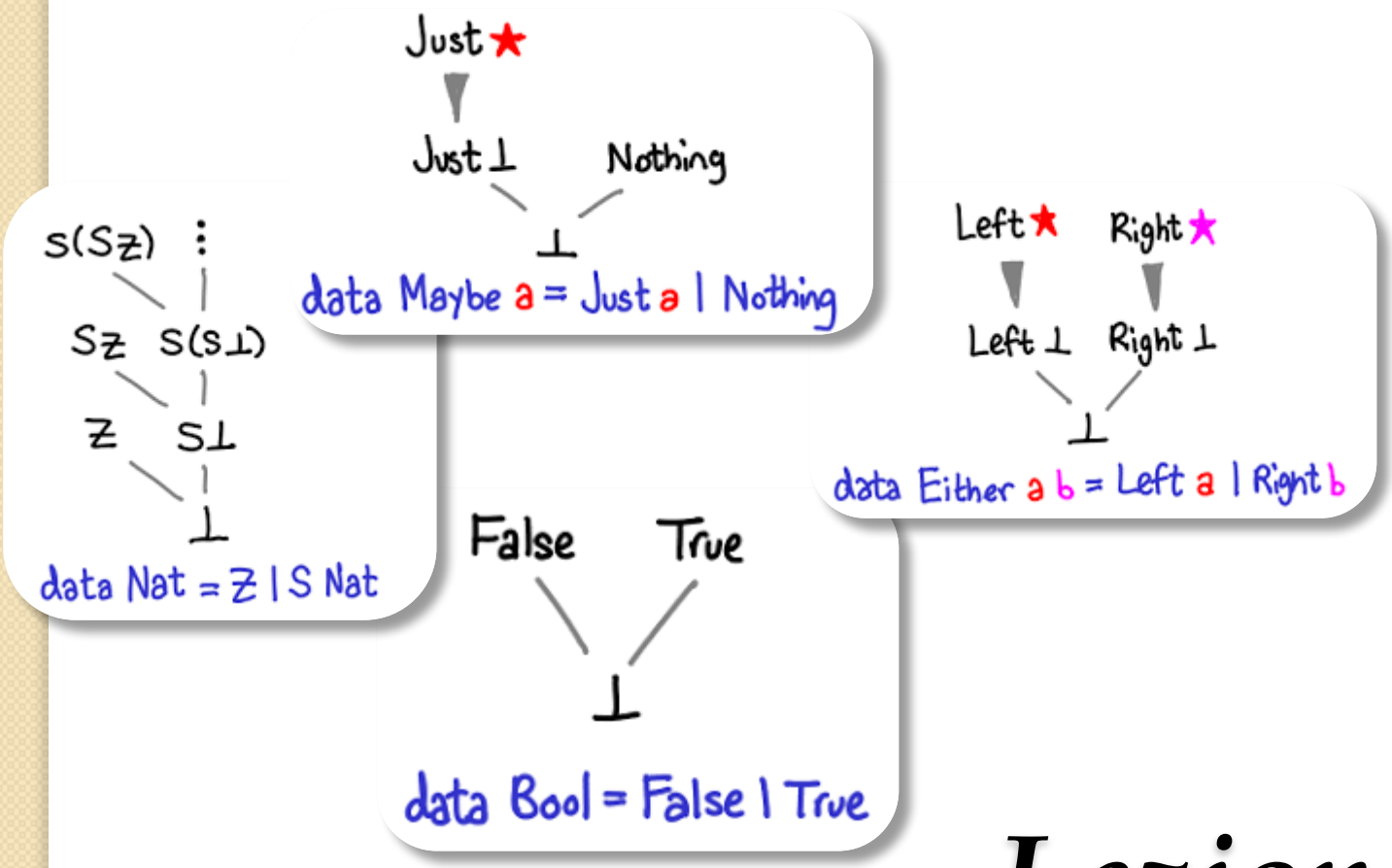
Tipi **Frazionari**, **`Fractional`** che definisce tipi numerici che hanno inoltre le seguenti funzioni (esempio **`Rational`** , **`Float`**):

```
recip, (/) :: a -> a -> a
```

```
 -- alcuni esempi con pi greco
>:t pi
Fractional a => a

>recip pi
0.3183098861837907

>toRational pi
884279719003555 % 281474976710656
```



Lezione 5b

Definizioni di Tipi di Dato in Haskell

Dichiarazioni di sinonimi di tipo

Con la parola chiave **type** possiamo definire **nuovi nomi** di **tipo** che sono essenzialmente **sinonimi** di espressioni di tipo.

Pur essendo 'solo' sinonimi possono essere utili per introdurre un livello di **astrazione** e/o **leggibilità** nei tipi delle funzioni.

Tuttavia **non offrono** nessun livello di **protezione** sui dati (**equivalenza strutturale**).

```
type Casella = (Int, Int)
type Move = (Int, Int)
 -- Posso dichiarare(=forzare) il tipo di una funzione move:
move :: Casella -> Move -> Casella
move (x,y)(dx,dy)=(x+dx, y+dy)
 -- ma anche:
move' (x,y)(dx,dy)=(x+dx, y+dy)
>:t move'
move'::(Num t1,Num t) => (t, t1) -> (t, t1) -> (t, t1)
 -- Casella, Posizione sono compatibili
move'' p p' = move' (move p p') (1,1)
>:t move'' -- notare i tipi e provare a spiegare!
move':: Casella -> Move -> (Int, Int)
```

Sinonimi parametrici

I sinonimi possono essere anche **parametrici**, cioè possono **dipendere da variabili di tipo**.

Vediamo due esempi e i **controlli** fatti dal compilatore Haskell:

```
 -- Posso rinominare il tipo coppia
type Pair a b = (a, b)
 -- oppure le coppie omogenee
type PairH a = (a, a)

maxP:: Ord a => PairH a -> a -- definisco il tipo
maxP (x, y) = if x > y then x else y
fstP:: PairH a -> a
fstP (x, y) = x
fstQ:: Pair a b -> a
fstQ (x, y) = x

 -- qualche controllo viene fatto, però:
>fstP (2, [2])
 -- dà errore, perché non è una PairH, ma...:
>fstQ (2, [2])
2
```

Dichiarazioni di nuovi tipi

Più interessante la definizione di **nuovi tipi** attraverso l'**uso di costruttori**. Ci sono numerosi, famosi, tipi finiti, ad esempio il tipo `Bool` o il tipo dei `SetteNani`.

Anche questi tipi possono dipendere da variabili di tipo.

`Either` è come dei `booleani parametrici`. **Esempio**: se rappresentate espressioni aritmetiche con alberi, un nodo potrebbe essere **`Either (Num a) (Num a => a -> a -> a)`**

```
 -- Booleani – con costruttori costanti
data Bool = False | True

 -- Sette Nani
data SetteNani = Eolo | Pisolo | Brontolo | …

 -- Giorni Settimana
data Giorni = Lun | Mar | Mer | Gio | Ven | Sab | Dom

 -- Più interessanti i tipi parametrici: coppie home made:
data Product a b = P a b

 -- oppure le somme disgiunte:
data Either a b = Lft a | Rgt b
```

Definizioni di funzioni

Le funzioni si possono sempre definire per **pattern matching**, esattamente come per i tipi predefiniti.

Significa che ho **gratis**, i **distruttori**, cioè la possibilità di `**decomporre**' in componenti valori di un tipo strutturato

```
 -- esempio con i sette nani
altezza Cucciolo =  99
altezza    _     = 124
 -- esempio con le coppie
myFst (P x y) =  x
mySnd (P x y) =  y
```

```
--Esempi di tipaggi e valutazioni:
> :t altezza
altezza :: Num a => SetteNani -> a
> altezza Cucciolo
99
> altezza Eolo
124
> :t myFst
myFst :: Pairs t1 t -> t
```

Definizioni di Istanze

C'è un meccanismo per dire che un tipo **è istanza di una classe**.

Ad esempio, il tipo booleano ammette uguaglianza e può dunque essere dichiarato istanza di **`Eq`** oppure di **`Ord`**.

Occorre contestualmente **fornire il codice di ==** mentre non è necessario fornire quello di **`/=`** in quanto **definito in modo standard** come la negazione di ==.

Similmente con <.

Solo i tipi definiti con **`data`** (e non con **`type`**) possono essere dichiarati istanze

```
 -- definizione di Bool come istanza di Eq
instance Eq Bool where
   False == False = True
   True  == True  = True
     _   ==   _   = False
 -- definizione di Bool come istanza di Ord
instance Eq Ord where
   False < True  = True
     _   <   _   = False
```

Derivazione da Classi

Ogni volta che si definisce un tipo, è buona norma considerare se dichiararlo (o meno) **istanza di qualche classe**, in particolare delle classi **predefinite**.

Per le classi `Eq, Ord, Show e Read` è possibile usare un **meccanismo automatico**, detto **derivazione**.

L'**uguaglianza è sintattica**, mentre l'**ordine** viene derivato (lessicograficamente) dall'**ordine** con cui si scrivono i **costruttori**.

La conversione in stringa prende i **nomi dei costruttori**.

```
 -- usare deriving per creare istanze
data Bool = False | True
          deriving (Eq, Ord, Show, Read)

> False < True
True
> show False
"False"
```

Altre Derivazioni da Classi

Facciamo un altro esempio con i giorni della settimana.

Posso definire la lista **lavorativi** perché ho dichiarato che **Giorni** deriva **Enum** (tipi enumerati con un'operazione di **succ**)

La **valutazione** di lavorativi **produce la stampa** della lista perché ho dichiarato **Giorni derivare Show**.

Un po' più complessa la classe **Read**: occorre dare informazione di tipo che non sempre può essere inferita dal compilatore.

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

-- (Lun) come fosse un operatore
lavorativi = [(Lun)..(Ven)]

> lavorativi
[Lun,Mar,Mer,Gio,Ven]

> g = read("Lun") :: Giorni
-- la stringa può venire da input
> g
Lun
```

```
-- succ sui giorni
> succ :: Enum a -> a -> a

> succ Lun
Mar

> succ Dom
*** Exception: succ{Giorni}:
tried to take `succ' of last
tag in enumeration
```

Costruttori di tipo: Maybe

Ecco il famoso tipo **Maybe** che è il tipo di **computazioni** che possono **fallire**. È il tipo con cui si rappresentano le **eccezioni**.

Un valore di Maybe può essere **Just v** (cioè un valore) oppure **Nothing** (computazione indefinita).

```
>:t Nothing
Nothing :: Maybe a
>:t Just
Just :: a -> Maybe a
>:t Just 'a'
Just 'a':: Maybe Char
-- usiamo eccezioni per trattare
-- funzioni parziali
safediv n 0 = Nothing
safediv n m = Just (n `div` m)
> :t safediv
Integral a => a->a->Maybe a
> safediv 3 0
Nothing
> safediv 7 3
Just 2
```

```
-- definizione Maybe
data Maybe a = Just a | Nothing
-- è un caso particolare di Either
```

```
-- funzione "inversa"
extract Just n = n
extract Nothing = undefined
> extract (safediv 3 0)
***Exception: Prelude.undefined
```

Tipi Induttivi o Ricorsivi

In Haskell si possono facilmente definire tipi **induttivi** o **ricorsivi** esattamente come visto finora, semplicemente **specificando la segnatura** (o tipo) **dei costruttori**.

Questo modo di procedere generalizza il tipo delle liste predefinite, costruite a partire da **lista vuota** `[]` e dalla funzione **cons** `(:)`.

```
 -- Naturali “unari”
data Nat = Zero | Succ Nat
 -- Liste
data List a = Nil | Cons a (List a)
 -- Alberi binari F=Foglia, R=radice
data BTree a = F a | R a (BTree a) (BTree a)
 -- o anche:
data BTree’a = Empty | Node (Btree’ a) (Btree’ a)
 -- anche gli amati lambda-termini:
data Lambda = Var Int | Apply Lambda Lambda |
      Abs Int Lambda
```

Definizioni di funzioni ricorsive

Le funzioni si possono sempre definire per **ricorsione** sui costruttsori, usando il **pattern matching**, esattamente come per le liste. Significa che ho gratis, i **distruttori**.

Vediamo la lunghezza e append tra liste.

E la visita in order che produce una lista.

```
 -- Lunghezza di una lista
length Nil = Zero
length (Cons _ l) = Succ (length l)
 -- append tra liste
append l Nil = l
append Nil l = l
append (Cons x l) m = Cons x (append l m)
-- visita inorder di un albero binario
flatten (Leaf x) = Cons x Nil
flatten (Node x t1 t2) =
   append (flatten t1) (Cons x (flatten t2))
```

Esempio: Espressioni

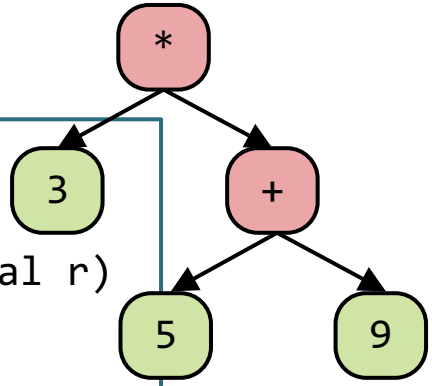
Facciamo un esempio di uso del tipo **`Either`**.

Immaginiamo di voler rappresentare un'**espressione aritmetica**: e usiamo come è naturale un albero.

In Haskell posso mettere le **funzioni nei nodi**! Ma alcuni nodi saranno operatori aritmetici, altri solo numeri. Quindi…

Ecco la funzione di valutazione expEval.

E la valutazione dell'espressione in figura.




```
type Exp a = BinTree (Either (a -> a -> a) (a))

expEval :: Num a => Exp a -> a
expEval (R (Left f) l r) = f (expEval l) (expEval r)
expEval (F (Right v))    = v

> btEval
   (R (Left (*))
      (F (Right 3))
      (R (Left (+)) (F (Right 9)) (F (Right 5))))
42
```

Esempio: Alberi

Vediamo in dettaglio cosa accade con la clausola **deriving**.

La deriving offre una stampa standard che riproduce la forma del termine usando i costruttori.

Oppure possiamo implementare noi un'alternativa migliore.

```
 -- Alberi binari F=Foglia, R=radice
data BTree a = F a | R a (BTree a) (BTree a)
   deriving Show

> R '*' (F '3') (R '+' (F '5') (F '9')) -- expr a caratteri
R '*' (F '3') (R '+' (F '5') (F '9'))

-- commentiamo deriving e mettiamo def esplicita di show
instance (Show a) => Show (BinTree a) where
   show (F v)      = show v
   show (R r d s) =  "("++show d ++ show r ++ show s ++")"
> r
('3''*'('5''+''9')) -- purtroppo show 'a'="'a'" 😔
```

Lezione 5

That's all Folks…

Grazie per l'attenzione…

…Domande?