Types & Functions

Functional Programming 2018

Christophe Scholliers

Based on Learn you a Haskell Chapter 3-6

Todo List

- Simple Types
- Composite Types
- Type Classes
- Pattern Matching
- Bindings
- Recursion
- Higher Order Functions
- Folding

Type Inference

```
*Main> :t 'a'
'a' :: Char
*Main> :t True
True :: Bool
*Main> :t 3 == 34
3 == 34 :: Bool
```

:: is read as "has type"

Composite types 1

```
Prelude> :t (‘a’,'b')
(‘a’,'b') :: (Char, Char)

Prelude> :t "hallo"
"hallo" :: [Char]
```

Haskell built-in Types

Simple Types	Composite Types
Int	[T]
Float	T1->T2
Char	(T1,T2)
Double	(T1,T2,T3,…)
…	…

Type Declaration

Argument

Return Type

```
removeNonUppercase :: [Char] -> [Char]
removeNonUppercase st = [ c | c <- st, c `elem` ['A'..'Z']]

addThree :: Int -> Int -> Int -> Int
addThree x y z = x + y + z
```

Optional but it is good coding style to provide type declarations for all top level functions

Multiple Arguments

Arguments

Return Type

```haskell
addThree :: Int -> Int -> Int -> Int
addThree x y z = x + y + z
```

Currying

```haskell
addThree :: Int -> (Int -> (Int -> Int))
addThree x y z = x + y + z
```

Number Types

```haskell
factorial :: Integer -> Integer
factorial n = product [1..n]


circumference :: Float -> Float
circumference r = 2 * pi * r


circumference' :: Double -> Double
circumference' r = 2 * pi * r
```

```
ghci> factorial 50
30414093201713378043612608
16606476884437764156896051
2000000000000

ghci> circumference 4.0
25.132742

ghci> circumference' 4.0
25.132741228718345
```

Type Variables

"generics"

```
Prelude> :t fst
fst :: (a, b) -> a
Prelude> :t snd
snd :: (a, b) -> b
```

fst and second are Polymorphic functions

Type Classes

Type Classes

```
Prelude> :t (==)
(==) :: Eq a => a -> a -> Bool
```

Class constraint

Type Classes

```
Prelude> :i Eq
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

Operations

Type Classes

Type classes != classes

Type classes ~ interfaces

Eq Type Class

"Types which can be tested for equality "

```
Prelude> 3.234 == 3.234
True

Prelude> 5 /= 243
True
```

```
(==) :: a -> a -> Bool
(/=) :: a -> a -> Bool
```

Ord Type Class

"Types which can be ordered"

```
Prelude> min 2 34
2

Prelude> 234 <= 345
True

Prelude> 234 >= 345
False

Prelude> "Hallo" < "Hello"
True
```

```
compare :: a -> a -> Ordering
(<) :: a -> a -> Bool
(<=) :: a -> a -> Bool
(>) :: a -> a -> Bool
(>=) :: a -> a -> Bool
max :: a -> a -> a
min :: a -> a -> a
```

Enum Type Class

“Types that can be enumerated”

```
succ :: a -> a
pred :: a -> a
toEnum :: Int -> a
fromEnum :: a -> Int
enumFrom :: a -> [a]
enumFromThen :: a -> a -> [a]
enumFromTo :: a -> a -> [a]
enumFromThenTo :: a -> a -> a -> [a]
```

```
Prelude> succ 3
4
Prelude> succ 'a'
'b'
```

Bounded Type Class

"Types that have an upper and lower bound"

```
minBound :: a
maxBound :: a
```

Polymorphic constant

```
Prelude> minBound :: Int
-9223372036854775808
Prelude> maxBound :: Int
9223372036854775807
```

Num Type Class

"Types that can be used as numbers"

```
(+) :: a -> a -> a
(-) :: a -> a -> a
(*) :: a -> a -> a
negate :: a -> a
abs :: a -> a
signum :: a -> a
```

```
Prelude> 30 :: Integer
30
Prelude> 30 :: Double
30.0
Prelude> :t (+)
(+) :: Num a => a -> a -> a
```

Integral Type Class

"Types of whole number types"

```
quot :: a -> a -> a
rem :: a -> a -> a
div :: a -> a -> a
mod :: a -> a -> a
quotRem :: a -> a -> (a, a)
divMod :: a -> a -> (a, a)
toInteger :: a -> Integer
```

```
Prelude> fromIntegral ( length [1,2,3])  + 2.3
5.3


Prelude>  length [1,2,3] + 2.3

<interactive>:18:19:
    No instance for (Fractional Int) arising from the literal ‘2.3’
    In the second argument of ‘(+)’, namely ‘2.3’
    In the expression: length [1, 2, 3] + 2.3
    In an equation for ‘it’: it = length [1, 2, 3] + 2.3
```

Show Type Class

"Types of whose values can be represented as a string"

```
showsPrec :: Int -> a -> ShowS
show :: a -> String
showList :: [a] -> ShowS
```

```
Prelude> show 2
"2"
Prelude> show True
"True"
Prelude> show [1,2,3,4]
"[1,2,3,4]"
```

Read Type Class

"Types whose values can be reconstructed from a string"

```
readsPrec :: Int -> ReadS a
readList :: ReadS [a]
```

```
Prelude> :t read
read :: Read a => String -> a
Prelude> read "23.23" + 23.2
46.43
Prelude> read "True"
*** Exception: Prelude.read: no parse
Prelude> read "True" :: Bool
True
```

Pattern Matching

Pattern matching

```
lucky :: (Integral a) => a -> String
lucky 7 = "LUCKY NUMBER SEVEN!"
lucky x = "Sorry, you're out of luck, pal!”

sayMe :: (Integral a) => a -> String
sayMe 1 = "One!"
sayMe 2 = "Two!"
sayMe 3 = "Three!"
sayMe 4 = "Four!"
sayMe 5 = "Five!"
sayMe x = "Not between 1 and 5"
```

Pattern matching

```haskell
factorial :: (Integral a) => a -> a
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Pattern matching

Does not stop

```
factorial :: (Integral a) => a -> a
factorial n = n * factorial (n - 1)
factorial 0 = 1
```

Pattern matching

Can fail

```haskell
charName :: Char -> String
charName 'a' = "Albert"
charName 'b' = "Broseph"
charName 'c' = "Cecil"
```

```
*Main> charName 's'
"*** Exception: test.hs:(30,1)-(32,22): Non-exhaustive patterns in function charName
```

Pattern matching

```haskell
addVectors :: (Num a) => (a, a) -> (a, a) -> (a, a)
addVectors a b = (fst a + fst b, snd a + snd b)
```

```haskell
addVectors :: (Num a) => (a, a) -> (a, a) -> (a, a)
addVectors (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)
```

Wildcards _

```haskell
first :: (a, b, c) -> a
first (x, _, _) = x

second :: (a, b, c) -> b
second (_, y, _) = y

third :: (a, b, c) -> c
third (_, _, z) = z
```

Even in list comprehensions

```
ghci> let xs = [(1,3), (4,3), (2,4), (5,3), (5,6), (3,1)]
ghci> [a+b | (a,b) <- xs]
[4,7,6,8,11,4]
```

Pattern matching Lists :

```
tell :: (Show a) => [a] -> String
tell [] = "The list is empty"
tell (x:[]) = "The list has one element: " ++ show x
tell (x:y:[]) = "The list has two elements: " ++ show x ++ " and " ++ show y
tell (x:y:_) = "This list is long. The first two elements are: " ++ show x ++ " and " ++ show y
```

As Patterns

```
capital :: String -> String
capital "" = "Empty string, whoops!"
capital all@(x:xs) = "The first letter of " ++ all ++ " is " ++ [x]
```

Keep a reference to the whole

Cases *expressions*

```haskell
head' :: [a] -> a
head' [] = error "No head for empty lists!"
head' (x:_) = x
```

```haskell
head' :: [a] -> a
head' xs = case xs of [] -> error "No head for empty lists!"
                      (x:_) -> x
```

Cases *expressions*

```haskell
describeList :: [a] -> String
describeList xs = "The list is " ++ case xs of []  -> "empty."
                                               [x] -> "a singleton list."
                                               xs  -> "a longer list."
```

```haskell
describeList :: [a] -> String
describeList xs = "The list is " ++ what xs
    where what [] = "empty."
          what [x] = "a singleton list."
          what xs = "a longer list."
```

Guards

Guards

```
bmiTell :: (RealFloat a) => a -> String
bmiTell bmi
    | bmi <= 18.5 = "You're underweight, you emo, you!"
    | bmi <= 25.0 = "You're supposedly normal. Pffft, I bet you're ugly!"
    | bmi <= 30.0 = "You're fat! Lose some weight, fatty!"
    | otherwise   = "You're a whale, congratulations!"
```

Guards

```haskell
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | weight / height ^ 2 <= 18.5 = "You're underweight, you emo, you!"
    | weight / height ^ 2 <= 25.0 = "You're supposedly normal. Pffft, I bet you're ugly!"
    | weight / height ^ 2 <= 30.0 = "You're fat! Lose some weight, fatty!"
    | otherwise                   = "You're a whale, congratulations!"
```

Bindings

Where

```haskell
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | bmi <= 18.5 = "You're underweight, you emo, you!"
    | bmi <= 25.0 = "You're supposedly normal. Pffft, I bet you're ugly!"
    | bmi <= 30.0 = "You're fat! Lose some weight, fatty!"
    | otherwise   = "You're a whale, congratulations!"
    where bmi = weight / height ^ 2
```

Where

```haskell
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | bmi <= skinny = "You're underweight, you emo, you!"
    | bmi <= normal = "You're supposedly normal. Pffft, I bet you're ugly!"
    | bmi <= fat    = "You're fat! Lose some weight, fatty!"
    | otherwise     = "You're a whale, congratulations!"
    where bmi = weight / height ^ 2
          skinny = 18.5
          normal = 25.0
          fat = 30.0
```

Where

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | bmi <= skinny = "You're underweight, you emo, you!"
    | bmi <= normal = "You're supposedly normal. Pffft, I bet you're ugly!"
    | bmi <= fat    = "You're fat! Lose some weight, fatty!"
    | otherwise     = "You're a whale, congratulations!"
     where bmi = weight / height ^ 2
           (skinny, normal, fat) = (18.5, 25.0, 30.0)
```

Where

```haskell
calcBmis :: (RealFloat a) => [(a, a)] -> [a]
calcBmis xs = [bmi w h | (w, h) <- xs]
    where bmi weight height = weight / height ^ 2
```

Let

```
cylinder :: (RealFloat a) => a -> a -> a
cylinder r h =
    let sideArea = 2 * pi * r * h
        topArea  = pi * r ^2
    in  sideArea + 2 * topArea
```

Watch out with indentation !

Let

```
ghci> (let a = 100;
           b = 200;
           c = 300 in a*b*c,
           let foo="Hey ";
               bar = "there!"
            in foo ++ bar)
(6000000,"Hey there!”)

ghci> (let (a,b,c) = (1,2,3) in a+b+c) * 100
600
```

Let vs Where

```
ghci> [if 5 > 3 then "Woo" else "Boo", if 'a' > 'b' then "Foo" else "Bar"]
["Woo", “Bar"]

ghci> 4 * (if 10 > 5 then 10 else 0) + 2
42

ghci> 4 * (let a = 9 in a + 1) + 2
42
```

Expression !

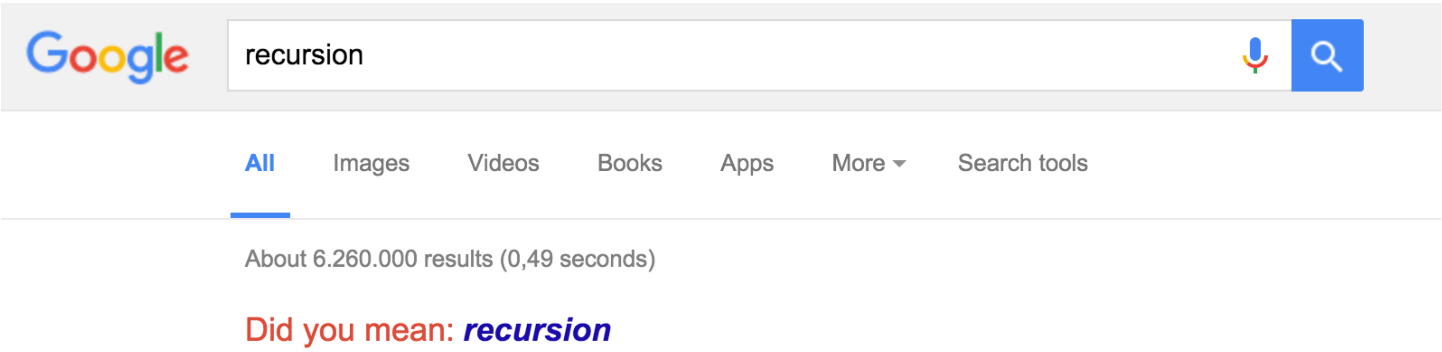
Different Lets !

Visible in the whole interaction

Only visible here

```
ghci> let zoot x y z = x * y + z
ghci> zoot 3 9 2
29
ghci> let boot x y z = x * y + z in boot 3 4 2
14
ghci> boot
<interactive>:1:0: Not in scope: `boot'
```

Recursion


recursion
All
Images
Videos
Books
Apps
More
Search tools
About 6.260.000 results (0,49 seconds)
Did you mean: recursion

Recursion

```haskell
maximum' :: (Ord a) => [a] -> a
maximum' [] = error "maximum of empty list"
maximum' [x] = x
maximum' (x:xs) = max x (maximum' xs)
```

maximum' [2,5,1] =
max 2 (maximum' [5,1] =
max 5 (maximum' [1] =
1))

Recursion

```haskell
replicate' :: (Num i, Ord i) => i -> a -> [a]
replicate' n x
    | n <= 0     = []
    | otherwise = x:replicate' (n-1) x
```

Recursion

```haskell
take' :: (Num i, Ord i) => i -> [a] -> [a]
take' n _
    | n <= 0   = []
take' _ []     = []
take' n (x:xs) = x : take' (n-1) xs
```

Recursion

```haskell
take' :: (Num i, Ord i) => i -> [a] -> [a]
take' n _
    | n <= 0   = []
take' _ []     = []
take' n (x:xs) = x : take' (n-1) xs
```

Higher Order Functions

```
applyTwice :: (a -> a) -> a -> a
applyTwice f x = f (f x)
```

```
ghci> applyTwice (+3) 10
16
```

Common operations

map

```haskell
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

```
ghci> map (+3) [1,5,3,1,6]
[4,8,6,4,9]
ghci> map (++ "!") ["BIFF", "BANG", "POW"]
["BIFF!","BANG!","POW!"]
ghci> map (replicate 3) [3..6]
[[3,3,3],[4,4,4],[5,5,5],[6,6,6]]
ghci> map (map (^2)) [[1,2],[3,4,5,6],[7,8]]
[[1,4],[9,16,25,36],[49,64]]
ghci> map fst [(1,2),(3,5),(6,3),(2,6),(2,5)]
[1,3,6,2,2]
```

filter

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
    | p x        = x : filter p xs
    | otherwise = filter p xs
```

```
ghci> filter (>3) [1,5,3,2,1,6,4,3,2,1]
[5,6,4]
ghci> filter (==3) [1,2,3,4,5]
[3]
ghci> filter even [1..10]
[2,4,6,8,10]
```

zip

```
Prelude> :t zip
zip :: [a] -> [b] -> [(a, b)]
```

```
Prelude> zip [1,2,3,4] "hallo"
[(1,'h'),(2,'a'),(3,'l'),(4,'l')]
```

unzip

```
Prelude> :t unzip
unzip :: [(a, b)] -> ([a], [b])
```

```
Prelude> unzip [('a','z'),('b','y'),('c','x'),('d','w')]
("abcd","zyxw")
```

zipWith

```
Prelude> :t zipWith
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

```
Prelude> zipWith (+) [1,2,3] [1..]
[2,4,6]
```

Quicksort revised

```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
    let smallerSorted = quicksort (filter (<=x) xs)
        biggerSorted  = quicksort (filter (>x)  xs)
    in  smallerSorted ++ [x] ++ biggerSorted
```

Finding a number

Find the largest number under 100,000 that's divisible by 3829

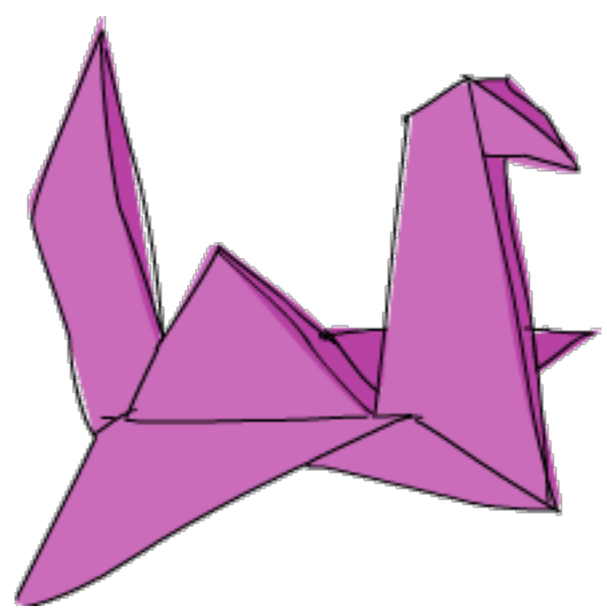
```haskell
largestDivisible :: (Integral a) => a
largestDivisible = head (filter p [100000,99999..])
    where p x = x `mod` 3829 == 0
```

Take While

```
Prelude> :t takeWhile
takeWhile :: (a -> Bool) -> [a] -> [a]
```

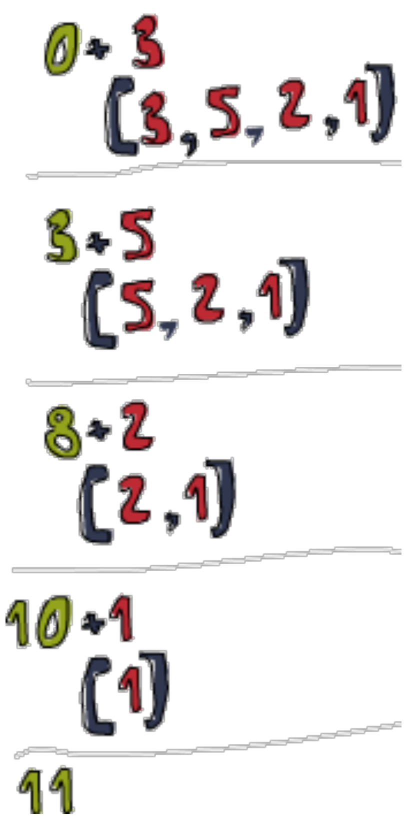
```
ghci> sum (takeWhile (<10000) (filter odd (map (^2) [1..])))
166650
```

Folding



foldl

```
Prelude> :t foldl
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
```




```
sum' :: (Num a) => [a] -> a
sum' xs = foldl (\acc x -> acc + x) 0 xs
```

```
ghci> sum' [3,5,2,1]
11
```

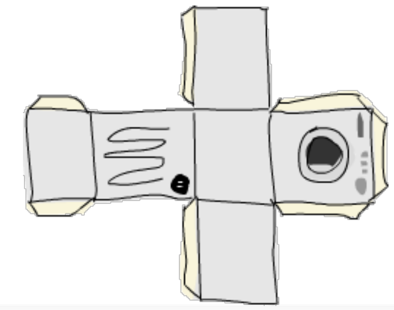
foldr

```
Prelude> :t foldr
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
```

```
sum' :: (Num a) => [a] -> a
sum' xs = foldr (\x acc -> acc + x) 0 xs
```

```
ghci> sum' [3,5,2,1]
11
```

Fold examples



```
maximum' :: (Ord a) => [a] -> a
maximum' = foldr1 (\x acc -> if x > acc then x else acc)

map' :: (a -> b) -> [a] -> [b]
map' f xs = foldr (\x acc -> f x : acc) [] xs

reverse' :: [a] -> [a]
reverse' = foldl (\acc x -> x : acc) []

product' :: (Num a) => [a] -> a
product' = foldr1 (*)

filter' :: (a -> Bool) -> [a] -> [a]
filter' p = foldr (\x acc -> if p x then x : acc else acc) []

head' :: [a] -> a
head' = foldr1 (\x _ -> x)

last' :: [a] -> a
last' = foldl1 (\_ x -> x)
```


THE End