

# Lesson 2

types and type classes in Haskell

A type is a collection of values

$\text{Bool} \rightarrow \{\text{True}, \text{False}\}$

$\text{Bool} \rightarrow \text{Bool} = \text{set of all functions from Bool to Bool}$

$\text{True} :: \text{Bool}$

$\text{not} :: \text{Bool} \rightarrow \text{Bool}$

for any expression  $e$ ,  $e :: T$

$\text{not} (\text{not False}) :: \text{Bool}$

type inference uses this rule

$$f :: A \rightarrow B \quad e :: A$$

---

$$f\ e :: B$$

not False :: Bool

not 3 error

in Haskell type inference is done at compile time, thus, before evaluating

hence Haskell is **type safe** in the strong sense that during evaluation, **no type error can occur**

in general, type checking can be done both statically and during evaluation

type safeness = all type errors are exposed (sooner or later)

don't fall in the trap of believing that in a type safe language no error can occur

type safeness assures that all type errors are exposed (but type errors can be done by programmers and other sort of errors can still occur)

this property is important !!

But there is something that may be viewed as a drawback:  
if True then 1 else False is not type correct in Haskell

static type safeness may be too restrictive (?)

if True then 1 else False

is not correct in Haskell because the two branches have different types

however, when it is executed only the then branch is used and thus the statement has just type Num

thus if we would execute it everything would be ok

but in Haskell this cannot be executed because the type check fails.

we may wonder whether this is really a restriction....

## Basic types

Bool, Char, String, Int, Integer, Float, Double

list types : a list is a asequence of values of the same type

[False,True,False] :: [Bool]

["one ", "two ", "three "] :: [String]

the type does not convey the length. We can have [[String]],  
[[[String]]] and so on..



# Tuples

`(1,2) :: (Num,Num)`

`(True, True, False) :: (Bool, Bool, Bool)`

`(True, 1, 'a', "one ") :: (Bool, Num, Char, String)`

`()`

the type of tuples conveys their arity

`('a', (False,'b')) :: (Char,(Bool,Char))`

`(['a','b'],('a',[True,False])) :: ([Char],(Char,[Bool]))`

# Functions

`not :: Bool -> Bool`

`add :: (Int , Int) -> Int`

`add (x,y) = x + y`

`zeroto :: Int -> [Int]`

`zeroto n = [0..n]`

function can be partial: `head []`

**\*\*\*Exception : Prelude.head: empty list**

# Curried functions

instead of :

`add :: (Int , Int) -> Int`

`add (x,y) = x + y`

generally in Haskell we use this:

`add' :: Int -> Int -> Int`

`add' x y = x + y`

this function is curried because it takes one parameter at the time

`add' :: Int -> (Int -> Int)`

`add' 2 = \x -> 2 + x`

$(\lambda x \rightarrow 2 + x) 3 = 2 + 3 = 5$

functions can have functions as results

functions can receive other functions as actual parameters

In such cases : Higher-order functions

functions are values as any other, like `Int` e `Bool`

`mult :: Int -> Int -> Int -> Int`

`mult x y z = x * y * z`

$\text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} = \text{Int} \rightarrow (\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}))$

and

$\text{mult } x \ y \ z = ((\text{mult } x) \ y) \ z$

we could also write:

$\text{mult} = \backslash x \rightarrow (\backslash y \rightarrow (\backslash z \rightarrow x * y * z))$

$\backslash = \text{lambda}$

we can write functions with no name

# Polymorphic types

`length :: [a] -> Int`

`a` is a type variable that stands for any type

`length` works for all lists, of any type !! Polymorphic

`id :: a -> a`

`head :: [a] -> a`

`take :: Int -> [a] -> [a]`

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

# Overloaded types

as usual `+`, `*` ecc are overloaded

```
> :t 1 + 2    :: Num a => a
```

```
> :t 1.1 + 2.0 :: Fractional a => a
```

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

```
> :t (*) :: Num a => a -> a -> a
```

all types with a type constraint like `Num a =>....` are overloaded types

# Type classes

as a type is a collection of values, a type class is a collection of types that support some overloaded operations (methods)

Haskell as some built-in classes:

**Eq** that contains types whose values can be compared for equality and inequality,

$(==) :: a \rightarrow a \rightarrow \text{Bool}$

$(\backslash=) :: a \rightarrow a \rightarrow \text{Bool}$

Bool, Char, Int, String Float.....are in Eq



also types `[a]` and `(a,b,c)` are in `Eq` as long as `a`, `b` and `c` are in `Eq`

```
[1,2] == [2,1]
```

```
False
```

```
('a',1) == ('a',1)
```

```
True
```

**Ord** that contains types that are instances of the equality class `Eq`, and, in addition, whose values are totally ordered and therefore possess the following methods:

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

```
(>=) :: a -> a -> Bool    min :: a -> a -> a    max :: a -> a -> a
```

all basic types are in Ord

also lists and tuples are in Ord, provided that their elements have type in Ord

lists, strings and tuples are ordered lexicographically

**Show** that contains all types whose values can be converted into strings of characters, by using this method:

`show :: a -> String`

All basic types are in Show, therefore:

`show True`

`"True "`

`show 'a'`

`"'a' "`

`show 34`

`"34 "`

**Read** is the dual of Show and contains all types whose values can be converted from strings using the method:

`read :: String -> a`

`read "False" :: Bool`

`False`

`read "[2,4,5]" :: [Int]`

`[1,2,3]`

`read "34" :: Int`

`34`

`read "('a',False)" :: (Char, Bool)`

`('a', False)`

the types in these expressions are mandatory to resolve the type of the result which, otherwise, would not be inferable

not `(read "False")` needs no type to be inferred

**Num** contains all numeric types. That is all types that can be processed with the following methods:

$(+) :: a \rightarrow a \rightarrow a$      $(-) :: a \rightarrow a \rightarrow a$      $(*) :: a \rightarrow a \rightarrow a$   
 $\text{negate} :: a \rightarrow a$      $\text{abs} :: a \rightarrow a$      $\text{signum} :: a \rightarrow a$

**Integral** contains types that are in Num that are integers and that support

$\text{div} :: a \rightarrow a \rightarrow a$   
 $\text{mod} :: a \rightarrow a \rightarrow a$

$\text{div } 2 \ 3$ , but often,  $2 \text{ 'div' } 3$

## Fractional

floating point types from Num that support,

$(/) :: a \rightarrow a \rightarrow a$

$\text{recip} :: a \rightarrow a$

## Exercises

1) type of  $([\text{True}, \text{True}], ['a', 'a'])$

2) type of  $[\text{tail}, \text{init}, \text{reverse}]$

3) write definitions that have the following types:

$\text{bools} :: [\text{Bool}]$ ,  $\text{nums} :: [[\text{Num}]]$ ,  $\text{add} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$

$\text{copy} :: a \rightarrow (a, a)$ ,  $\text{apply} :: (a \rightarrow b) \rightarrow a \rightarrow b$

4) type of following functions:

$\text{swap } (x, y) = (y, x)$ ,  $\text{second } xs = \text{head } (\text{tail } xs)$ ,  $\text{pair } x \ y = (x, y)$ ,

palindrome xs = reverse xs == xs

twice f x = f (f x)

functional types cannot be in Eq