Lesson 2

types and type classes in Haskell

A type is a collection of values

Bool -> {True, False}

Bool -> Bool = set of all functions from Bool to Bool

True :: Bool
not :: Bool -> Bool

for any expression e, e :: T
not (not False) :: Bool

type inference uses (also) this rule

$$\frac{f :: A \to B \quad e :: A}{f\ e :: B}$$

not False :: Bool
not 3 error

in Haskell type inference is done at compile time, thus, before the program is executed, we say that it is done statically

hence Haskell is type safe in the (strong) sense that during evaluation, no type error can occur
also called «strongly typed»

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 beleiving 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 sequence 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 (eterogeneous)

(1,2) :: (Num,Num)
(True, True, False) :: (Bool, Bool, Bool)
(True, 1, ‘a’, ″one ″) :: (Bool, Num, Char, String)
() :: () --unit
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

reverse :: [a]->[a]
reverse [] = []
reverse (x:xs) = (reverse xs) ++ [x]

zeroto :: Int ->[Int]
zeroto n = [0..n]

function can be partial:     head []
***Exception : Prelude.head: empty list
```

functions can have functions as results

functions can receive other functions as actual parameters

In such cases they are : higher-order functions

functions are values as any other, like Int and Bool

Currified functions

in place of :
add :: (int , Int) -> Int
add (x,y) = x + y

in Haskell + is currified:
:t (+)
(+) :: Num a => a -> a -> a = a -> (a -> a)

(+) 2 :: Num a => a -> a

currified is the default type for functions in Haskell

but be careful !

```
add (x,y) = x +y
add :: Num a => (a,a) -> a
```

in a way this type is also currrified but it takes immediately a pair of Num

lambda-expressions (from Church and Curry) to define functions

\ a b -> a + b*3 currified

formal
parameters body

with lambda we can define function without name that can be immediately called

or we can give them a name: f = \ a b -> a + b*3

```
mult x y z = x * y  * z
mult :: Num a => a -> a -> a -> a


mult = \ x ->  \y -> \z -> (x * y *z)

mult 2 = \y -> \z -> (2 * y *z)

mult 2  3= \z -> (2 * 3 *z)

mult 2 3 4 = 2 * 3 * 4
```

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 costraint like Num a =>.... are overloaded types

An example of an higher-order function:

```
f = (2+) . (3*)   -- (.) = function composition

f 2
8

f :: Num a => a -> a
(.) :: (b -> c) -> (a -> b) -> a -> c
```

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 has many built-in classes: Eq, Ord, Show, Read, Num, Integral, Fractional,....

Eq that contains types whose values can be compared for equality and inequality,
(==) :: a -> a -> Bool
(\=) :: a -> a -> Bool

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 wwhose values can be converted into strings of characters, by using this method:
show :: a -> String
All basic types are in Show, therefore:

show True	show ‘a’	show 34
″True ″	″’a’ ″	″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 inferrable
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 -> a -> a (-) :: a -> a -> a (*) :: a -> a -> a

negate :: a -> a abs :: a -> a signum :: a -> a

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

Int and Integer

div :: a -> a -> a

mod :: a -> a -> a

div 2 3, but often, 2 ‘div’ 3

Fractional

floating point types from Num that support,

(/) :: a -> a -> a

recip :: a -> a

Float and Double

Exercises

1) type of ([True, True], [‘a’,’a’])
2) type of [tail, init, reverse]
3) write definitions that have the following types:

bools :: [Bool], nums :: [[Num]], add :: Int ->Int -> Int -> Int

copy :: a -> (a,a), apply :: (a->b)->a -> b

4) type of following functions:

swap (x,y) = (y,x), second xs = head (tail xs), pair x y = (x,y),

palindrome xs = reverse xs == xs

twice f x = f (f x)

Don’t forget the type class constraints!

functional types cannot be in Eq

Homework 1

https://www.seas.upenn.edu/~cis194/spring13/hw/01-intro.pdf