Haskell Overview III (3A)

Young Won Lim 9/27/16 Copyright (c) 2016 Young W. Lim.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using OpenOffice.

Young Won Lim 9/27/16 Haskell Tutorial, Medak & Navratil ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf

Yet Another Haskell Tutorial, Daume https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf Prelude> 7 :: Int 7 Prelude> 7 :: Double 7.0

usually don't have to declare types (type inference) to declare types, use :: to do it.

4

Type Information

Prelude> :t False

False :: Bool

Prelude> :t 'A'

'A' :: Char

Prelude> :t "Hello, world"

"Hello, world" :: [Char]

Print type information

Type Classes

Prelude> :t 42 42 :: (Num t) => t Prelude> :t 42.0 42.0 :: (Fractional t) => t Prelude> :t gcd 15 20 gcd 15 20 :: (Integral t) => t

class constraint

42 can be used as any numeric type42.0 can be any fractional typegcd 15 20can be any integral type

the <u>type t is constrained</u> by the <u>context</u> (Num t), (Fractional t), (Integral t)

the **types** of t must *belong* to the Num / Factional / Integra **type class**

Instances

Num instances

Integral instances

Int	an integer with at least 30 bits of precision.	
Integer	an integer with <i>unlimited</i> precision.	
Float	a single precision floating point number.	
Double	a double precision floating point number.	
Rational	a fraction type, with no rounding error.	
Fractional instances		

7

a type class definition:

specifying a set of **functions** or **constants**, together with their respective types, Like the Interface in Java

that must be <u>implemented</u> for every type that is *belonged* to the **type class**

Type Class Definition

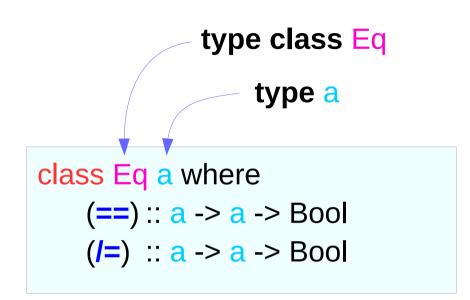
the **type class** Eq is intended to *contain* **types** that have implementations of equality (==), (/=) functions

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

a **type a** has an **instance** of the **class Eq** if there is an (overloaded) operation == and *I*= defined.

a **type** a *belongs* to the **type class** Eq if (==) and (*I*=) <u>functions are defined</u>

Instance of a Class



a **type** a can be an **instance** of the **class Eq** if there is an (overloaded) operation **==** and **/=** defined.

The **type** Integer is an **instance** of the **class** Eq, whose **method** == is defined

The **type** Float is an **instance** of the **class** Eq, whose **method** == is defined

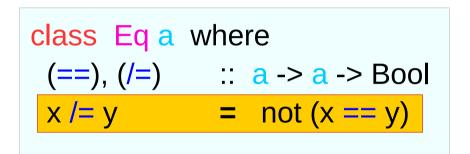
Instance Declaration

class Eq a where (==) :: a -> a -> Bool	type class Eq	type a
instance Eq Integer where x == y = $x integerEq y$	type class Eq Eq	instance Integer Float
instance Eq Float where		

= x `floatEq` y

https://en.wikipedia.org/wiki/Type_class

x == y



If a method is not defined in an instance declaration, then the default implementation defined in the class declaration, if it exists, is used instead.

overloaded method definition

The default definition can be overloaded in an instance declaration

Class Constraint

elem :: a -> [a] -> Bool

the function elem has the <u>type</u> a -> [a] -> Bool

the <u>type</u> a is *constrained* by the <u>context</u> (Eq a)

the **types** of a must *belong* to the Eq **type class**

=> : called as a 'class constraint'

Class Constraint Example

elem function definition which determines if an element is in a list

elem :: (Eq a) => a -> [a] -> Bool elem y [] = False elem y (x:xs) = (x == y) || elem y xs

Renaming module imports.

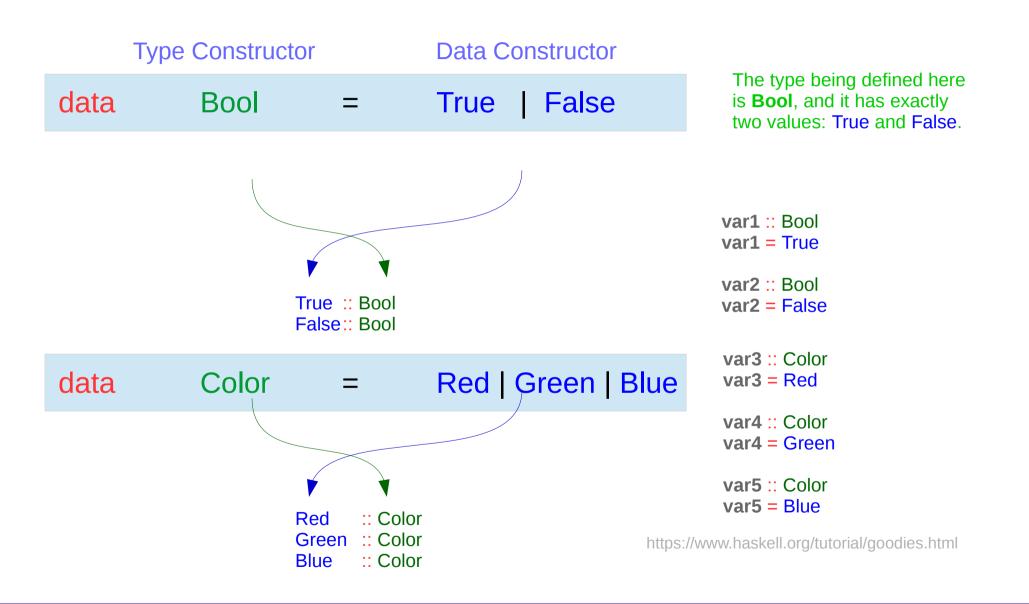
Like qualified and hiding, as is not a reserved word but may be used as function or variable name.

import qualified Data.Map as M

main = print (M.empty :: M.Map Int ())

https://wiki.haskell.org/Keywords#as

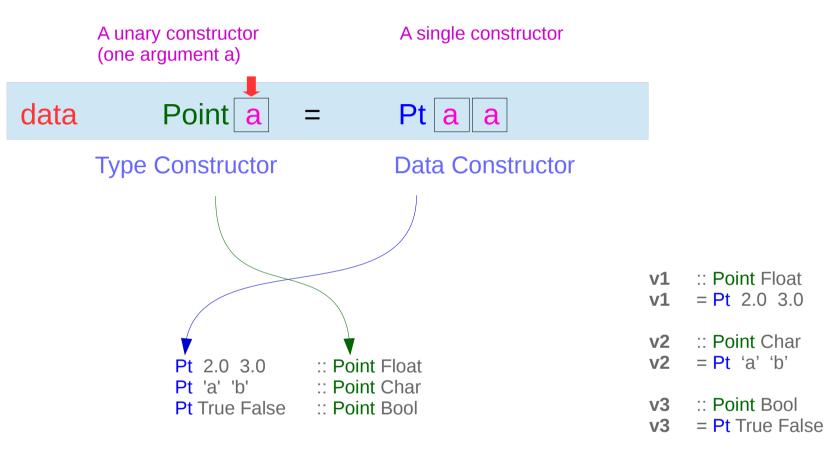
Enumerated Data Types



Type Names and Constructor Functions

A nullary constructor: takes no arguments		A multi-constructor
data	a Bool =	True False
Type Constructor		Data Constructor
	Type name : Bool The name of new data type	Constructor function : True, False
	Usually it appears in the linea concerning type information (::)	Usually it appears in the lines concerning application (=)

Parameterized Data Type Definition



Pt :: a -> a -> Point a

Solving a list of quadratic equations

```
roots :: (Float, Float, Float) -> (Float, Float)
roots (a,b,c) = if d < 0 then error "sorry" else (x1, x2)
where x1 = e + \text{sqrt d} / (2 * a)
x2 = e - \text{sqrt d} / (2 * a)
d = b * b - 4 * a * c
e = -b / (2 * a)
```

```
real ::(Float, Float, Float) -> Boolreal(a,b,c)= (b*b - 4*a*c) >= 0
```

```
p1 = (1.0,2.0,1.0) :: (Float, Float, Float)

p2 = (1.0,1.0,1.0) :: (Float, Float, Float)

ps = [p1,p2]

newPs = filter real ps

rootsOfPs = map roots newPs
```

data Polynom = Poly Float Float Float

Poly :: Float -> Float -> Polynom

- data the keyword
- Polynom the name of the data type
- Poly the constructor function (:t Poly)
- Float the three arguments to the Poly constructor

data Polynom = Poly Float Float Float



- p1, p2 :: Polynom
- p1 = Poly 1.0, 2.0, 3.0
- p2 = Poly 1.0, 3.0, (-5.0)

Recursive Data Type Example (1)

```
data Bus = Start | Next (Bus) deriving Show
```

myBus A = Start

```
myBusB = Next (Next (Next (Start)))
```

myBusC = Next myBusB

```
plus :: Bus -> Bus -> Bus
```

plus a Start = a

```
plus a (Next b) = Next (plus a b)
```

testBus :: Bus

```
testBus = plus myBusC myBusB
```

(Next b) pattern matching

howFar :: Bus -> Int

howFar Start = 0

howFar (Next r) = 1 + howFar r

(Next r) pattern matching

testInt :: Int

testInt = (+) (howFar myBusC) (howFar myBusB)

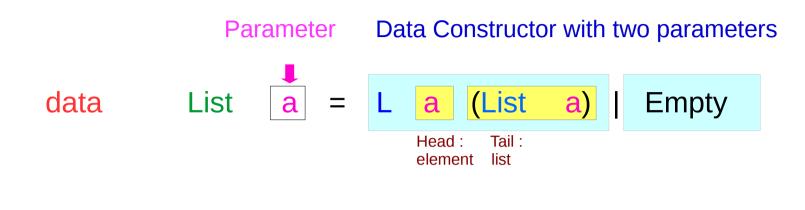


Any type is ok but The type of every element in the list must be the same

List = [] | (a : List)

an empty	a list with at least
list	one element
[]	(x:xs)

Parameterized Data Types



L1, L2, L3 :: List Integer L1 = Empty L2 = L 1 L1 L3 = L 5 L2 L4 = L 1.5 Empty :: List Double Constructor a (a)

Polymorphic Type

types that are universally quantified in some way over all types essentially describe families of types

(forall a) [a] is the family of types consisting of, for every type a, the type of lists of a.

- lists of integers (e.g. [1,2,3])
- lists of characters (['a','b','c'])
- lists of lists of integers, etc.
- [2,'b'] is not a valid example

Class Show

the instances of Show are those types that can be converted to character strings. (information about the class)

The function show

show :: (Show a) => a -> String

Similar to the toString() method in Java

data Tree a = Leaf a | Branch (Tree a) (Tree a)

Constructor Definitions

- Branch :: Tree a -> Tree a -> Tree a
- Leaf :: a -> Tree a

Eq Instance of Tree Type

Eq Instance

instance (Eq a) => Eq (Tree a) where
(Leaf x) == (Leaf y) =
$$x == y$$

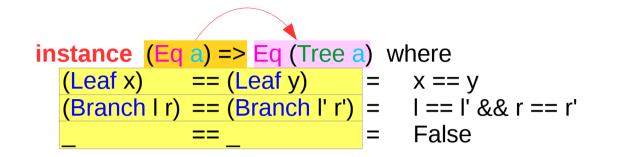
(Branch I r) == (Branch I' r') = $I == I' \&\& r == r'$
= False

The **type** Integer is an **instance** of the **class** Eq, whose **method** == is defined

The **type** Float is an **instance** of the **class** Eq, whose **method** == is defined

https://www.haskell.org/tutorial/stdclasses.html

Derived Instances



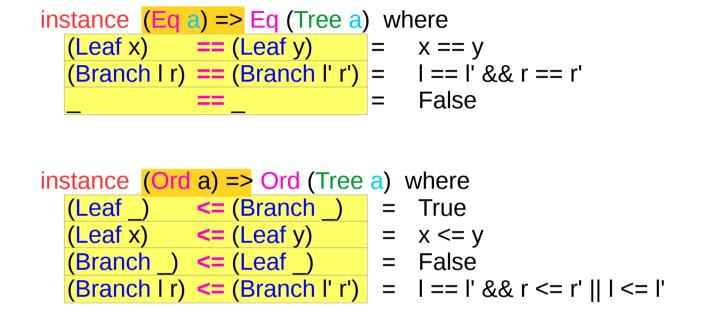
Automatically Derived Eq Instance

data Tree a = Leaf a | Branch (Tree a) (Tree a) deriving Eq



https://www.haskell.org/tutorial/stdclasses.html

Derived Instances



data Tree a = Leaf a | Branch (Tree a) (Tree a) deriving (Eq, Ord)

data [a] = [] | a : [a] **deriving** (Eq, Ord)

https://www.haskell.org/tutorial/stdclasses.html

Haskell Overview III

 $\begin{array}{ll} \mbox{data T0 f a = MkT0 a} & \mbox{deriving (Eq)} \\ \mbox{data T1 f a = MkT1 (f a)} & \mbox{deriving (Eq)} \\ \mbox{data T2 f a = MkT2 (f (f a)) deriving (Eq)} \end{array}$

instance Eq a=>Eq (T0 f a) where ...instance Eq (f a)=>Eq (T1 f a) where ...instance Eq (f (f a))=>Eq (T2 f a) where ...

Similar to the toString() method in Java

roots :: (Floating a) => $(a, a, a) \rightarrow (a, a)$

Parameterized Polymorphism

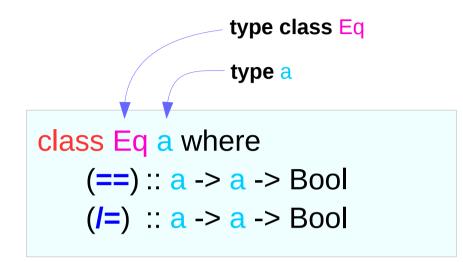
- plus :: a -> a -> a,
- plus :: Int -> Int -> Int,
- plus :: Rat -> Rat -> Rat,

data List a = L a (List a) | Empty

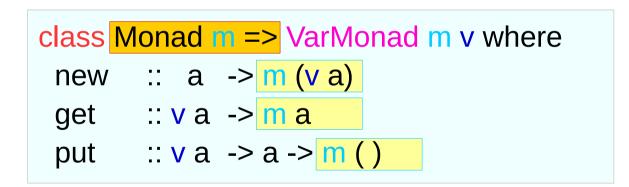
listlen :: List a -> Int listlen Empty = 0 listlen (L _ list) = 1 + listlen list

(L _ list) pattern matching

Multi-parameter Type Class Definition



SPTC: a type class is a set of types



MPTC: a type class is a relation between types

https://wiki.haskell.org/Multi-parameter_type_class

Multi-parameter Type Class Definition

class Monad m => VarMonad m v where				
new	:: a -> <mark>m (v a)</mark>			
get	::va -> <mark>ma</mark>			
put	::va ->a-> <mark>m()</mark>			

instance VarMonad IO IORef where ... instance VarMonad (ST s) (STRef s) where ...

{-# LANGUAGE MultiParamTypeClasses #-} pragma

https://wiki.haskell.org/Multi-parameter_type_class

type ID = Int type Attrib = (String, String)

class Objects o where object :: ID -> [Attrib] -> o getID :: o -> ID getAttr :: o -> [Attrib] getName :: o -> String getName = snd . head . filter (("name"==) . fst) . getAttr

class (Object o) => Databases d o where

```
empty:: d ogetLastID:: d o -> IDgetObjects:: d o -> [o]setLastID:: ID -> d o -> d osetObjects:: [o] -> d o -> d o
```

```
insert :: [Attrib] -> d o -> d o
insert as db = setLastID i' db' where
    db' = setObjects os' db
    os' = 0 : os
    os = getObjects db
    o = object i' as
    i' = 1 + getLastID db
```

```
select :: ID -> d o -> o
select i = head . filter ((i==).getID) . GetObjects
```

```
selectBy :: (o -> Bool) -> d o -> [o]
selectBy f = filter f . getObjects
```

data Object = Obj ID [Attrib] deriving Show

```
instance Objects Object
```

object i as = Obj i as getID (Obj i as) = i getAtts (Obj i as) = as

data DBS o = DB ID [o] deriving Show

```
instance Databases DBS Object where
Empty = DB 0 []
getLastID (DB i os) = i
setLastID I (DB j os) = DB i os
getObjects (DB i os) = os
setObjects os (DB i ps) = DB i os
```

```
d0, d1, d2 :: DBS Object
d0 = empty
d1 = insert [("name", "john"), ("age", "30")] d0
d2 = insert [("name", "mary"), ("age", "20")] d1
test1 :: Object
test1 = select 1 d1
test2 :: Object
test2 = selectBy (("john" ==).getName) d2
```