Haskell Overview III (3A)

Young Won Lim
9/27/16

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

This document was produced by using OpenOffice.

Based on

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

Type Inference

```
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.

https://wiki.haskell.org/Learn_Haskell_in_10_minutes

Type Information

```
Prelude> :t False
False :: Bool
Prelude> :t 'A'
'A' :: Char
Prelude> :t "Hello, world"
"Hello, world" :: [Char]
```

Print type information

https://wiki.haskell.org/Learn_Haskell_in_10_minutes

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

42 can be used as any numeric type

42.0 can be any fractional type

gcd 15 20can be any integral type

class constraint

the type t is *constrained* by
the context (Num t),
(Fractional t), (Integral t)

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

https://wiki.haskell.org/Learn_Haskell_in_10_minutes

Instances

Num instances

Integral instances

Int	an integer with at least *30 bits* of precision.
Integer	an integer with *unlimited* 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

https://wiki.haskell.org/Learn_Haskell_in_10_minutes

Type Class

a type class definition:

specifying
a set of
functions or
constants,
together with their respective types,

Like the Interface in Java

that must be implemented
for every type that is *belonged* to the **type class**

https://en.wikipedia.org/wiki/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 ***/=*** defined.

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

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

Instance of a Class

type class Eq

type a

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

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

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

Instance Declaration

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

```
instance Eq Integer where
  x == y    =  x `integerEq` y
```

```
instance Eq Float where
  x == y    =  x `floatEq` y
```

type class	type
Eq	a

type class	instance
Eq	Integer
Eq	Float

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

Default Method

```
class  Eq a  where
  (==), (/=)        ::  a -> a -> Bool
  x /= y            =   not (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

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

Class Constraint

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

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

```
elem :: (Eq a) => 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'

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

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
```

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

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 Constructor — Data Constructor

```
data Bool = True | False
```

The type being defined here is **Bool**, and it has exactly two values: True and False.

```
True  :: Bool
False :: Bool
```

```
var1 :: Bool
var1 = True

var2 :: Bool
var2 = False
```

```
data Color = Red | Green | Blue
```

```
Red   :: Color
Green :: Color
Blue  :: Color
```

```
var3 :: Color
var3 = Red

var4 :: Color
var4 = Green

var5 :: Color
var5 = Blue
```

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

Type Names and Constructor Functions

A nullary constructor:
takes no arguments

A multi-constructor

```
data        Bool        =        True  | False
```

Type Constructor

Data Constructor

Type name : Bool
The name of new data type

Usually it appears in the linea concerning type information (**::**)

Constructor function : True, False

Usually it appears in the lines concerning application (**=**)

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

Parameterized Data Type Definition

A unary constructor
(one argument a)

A single constructor

data Point a = Pt a a

Type Constructor

Data Constructor

Pt 2.0 3.0 :: Point Float
Pt 'a' 'b' :: Point Char
Pt True False :: Point Bool

Pt :: a -> a -> Point a

v1 :: Point Float
v1 = Pt 2.0 3.0

v2 :: Point Char
v2 = Pt ‘a’ ‘b’

v3 :: Point Bool
v3 = Pt True False

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

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 + sqrt d / (2 * a)
            x2 = e - sqrt d / (2 * a)
            d = b * b - 4 * a * c
            e = - b / (2 * a)

real ::  (Float, Float, Float) -> Bool
real     (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
```

User defined type example (1)

```
data    Polynom    =  Poly    Float Float Float
```

Poly :: Float -> 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

User defined type example (2)

```
data     Polynom     =   Poly    Float Float Float
```

```
roots' :: Float Float Float -> (Float, Float)
roots' a b c        = … function definition …
```

```
roots2  :: Polynom -> (Float, Float)
roots2  (Poly a b c) = … function definition …
```

(Poly a b c) **pattern matching**

```
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**

Recursive Data Type Example (2)

```
howFar :: Bus -> Int
howFar Start = 0
howFar (Next r) = 1 + howFar r
```

(Next r) **pattern matching**

```
testInt :: Int
testInt = (+) (howFar myBusC) (howFar myBusB)
```

Recursive Definition of Lists

data [a] = [ ] | a : [a]

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

List = [ ] | (a : List)

an empty list

a list with at least one element

[ ]

(x:xs)

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

Parameterized Data Types

Parameter   Data Constructor with two parameters

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

Head : element   Tail : list

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

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

Show Class

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

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

Recursive Definition of Tree

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

Constructor Definitions

```
Branch  :: Tree a -> Tree a -> Tree a

Leaf    :: a -> Tree a
```

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

Eq Instance of Tree Type

Eq Instance

```
instance (Eq a) => Eq (Tree a) where
    (Leaf x)       == (Leaf y)       =   x == y
    (Branch l r)   == (Branch l' r') =   l == l' && r == r'
    _              == _              =   False
```

```
instance Eq Integer where
  x == y    =  x `integerEq` y
```

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

```
instance Eq Float where
  x == y    =  x `floatEq` y
```

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

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

Derived Instances

```
instance (Eq a) => Eq (Tree a) where
    (Leaf x)       == (Leaf y)         =    x == y
    (Branch l r)   == (Branch l' r')   =    l == l' && r == r'
    _              == _                =    False
```

Automatically Derived ***Eq Instance***

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

Eq a → Eq (Tree a)

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

Derived Instances

```
instance  (Eq a) => Eq (Tree a)  where
    (Leaf x)        == (Leaf y)         =     x == y
    (Branch l r)  == (Branch l' r')  =     l == l' && r == r'
    _                   == _                     =     False
```

```
instance  (Ord a) => Ord (Tree a)  where
    (Leaf _)          <= (Branch _)       =    True
    (Leaf x)          <= (Leaf y)           =    x <= y
    (Branch _)      <= (Leaf _)           =    False
    (Branch l r)    <= (Branch l' r')    =    l == l' && r <= r' || l <= l'
```

```
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

Deriving

```
data T0 f a = MkT0 a         deriving ( Eq )
data T1 f a = MkT1 (f a)     deriving ( Eq )
data T2 f a = MkT2 (f (f a)) deriving ( Eq )
```

```
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

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

Subset Polymorphism

```
roots    ::    (Floating a)   =>   (a, a, a)  ->   (a, a)
```

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

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**

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

Multi-parameter Type Class Definition

type class Eq

type a

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

SPTC: a type class is a set of types

```
class Monad m => VarMonad m v where
  new     ::   a   -> m (v a)
  get     :: v a  -> m a
  put     :: v a  -> a -> m ( )
```

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   -> m (v a)
  get    :: v a  -> m a
  put    :: v a  -> a -> m ( )
```

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

{-# LANGUAGE MultiParamTypeClasses #-} pragma

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

A Simple Database

```
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
```

A Simple Database

```
class (Object o) => Databases d o where
      empty         :: d o
      getLastID     :: d o -> ID
      getObjects    :: d o -> [o]
      setLastID     :: ID -> d o -> d o
      setObjects    :: [o] -> d o -> d o

      insert :: [Attrib] -> d o -> d o
      insert as db = setLastID i’ db’ where
              db’ = setObjects os’ db
              os’ = o : 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
```

A Simple Database

```
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
```

A Simple Database

```
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
```