FUNCTIONAL PROGRAMMING
NO.9 TYPE AND CLASS

Tatsuya Hagino
hagino@sfc.keio.ac.jp

Slide URL

https://vu5.sfc.keio.ac.jp/slide/

Static Type Checking and Type Inference

- Type
  - a set of values
    - Bool = { True, False }
    - Char = { 'a', 'b', ... }
    - Int = { ... -2, -1, 0, 1, 2, 3, ... }

- Static type checking
  - Haskell checks type of each expression when compile.
  - static = compile time (v.s. dynamic = run time)
  - Each expression has a proper type.

- Type inference
  - Haskell tries to infer type of a given expression.
  - Type inference may fail when information is not enough.

```
main = print f

f = read "80"
```

```
main = print f

f::Int
f = read "80"
```

Type Declaration

var$_1$, ***var***$_2$, ..., ***var***$_n$ :: ***type***

- Explicitly declaring type of variables
  - Help type inference
  - Express your intention ⇒ Easy to debug

```
defaultLines::Int
ul, ol, li::String -> String
```

- Type declaration of an expression
  - Declare type of an expression inside an expression

```
luckyNumber = (7 :: Int)
unluckyNumber = (13 :: Integer)
```

Polymorphic Type

- Type may have type variables

- Polymorphic type
  - type with type variables

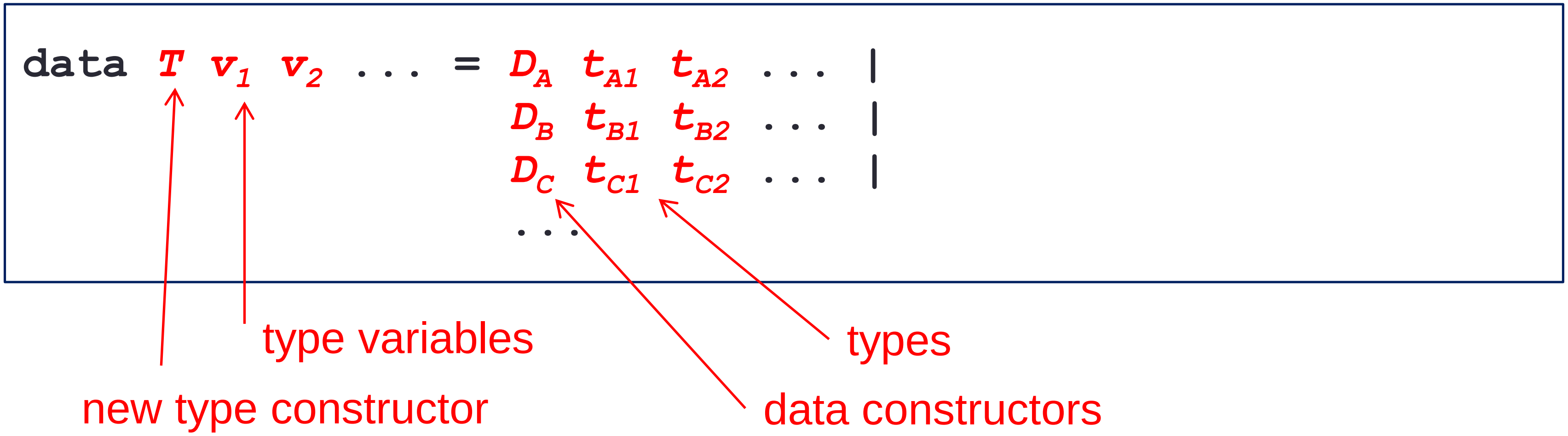
```
length :: [a] -> Int

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

- Type variables can be instantiated to any types.

Algebraic Data Type

- New type can be declared using data declaration.



$$
\begin{aligned}
\texttt{data}\ T\ v_1\ v_2\ \ldots = {} & D_A\ t_{A1}\ t_{A2}\ \ldots\ | \\
& D_B\ t_{B1}\ t_{B2}\ \ldots\ | \\
& D_C\ t_{C1}\ t_{C2}\ \ldots\ | \\
& \ldots
\end{aligned}
$$

type variables

types

new type constructor

data constructors

- A value of type ***T*** can be created by D_A, D_B, D_C,...
- Type name and data construct name need to start with a capital letter.

Example

```
data Anchor = A String String
```

- A new type `Anchor` is declared.

- `A` is the data constructor of type `Anchor`.
  - `A` has two String fields.
  - `A :: String -> String -> Anchor`

```
href = A "http://www.sfc.keio.ac.jp/" "SFC Home Page"
```

- Use data construct pattern to access fields.

```
compileAnchor (A url label) = ...
```

Field Label

- Provide label to fields of data constructors.

```
data Anchor = A { aURL :: String, aLabel :: String }
```

- Labels can be used in data construct patters to access fields.

```
compileAnchor (A { aURL = u, aLabel = l }) = ...

anchorUrl (A { aURL = u }) = u
```

- Field labels can be used as selectors to access fields.
  - `aURL :: Anchor -> String`
  - `aLabel :: Anchor -> String`

```
href = A "http://www.sfc.keio.ac.jp/" "SFC Home Page"

main = do print (aLabel href)
```

Field Label (cont.)

- Field label can be used to create a new value by changing some fields of an existing value.

```
data Anchor = A { aURL :: String, aLabel :: String }

href = A "http://www.sfc.keio.ac.jp/" "SFC Home Page"

main = do print href
          print (href { aLabel = "that" })
```

outputs `A "http://www.sfc.keio.ac.jp/" "SFC Home Page"`

outputs `A "http://www.sfc.keio.ac.jp/" "that"`

Creating Polymorphic Data Type

- Use type variable to create polymorphic data type

```
data Stack a = MkStack [a]
```

type variable

```
MkStack [True, False]    -- Stack Bool

MkStack ['a', 'b', 'c'] -- Stack Char

MkStack ["aa", "bb"]     -- Stack String
```

Enumeration Type

- Use | to create enumeration type

```
data OpenMode = ReadOnly | WriteOnly | ReadWrite
```

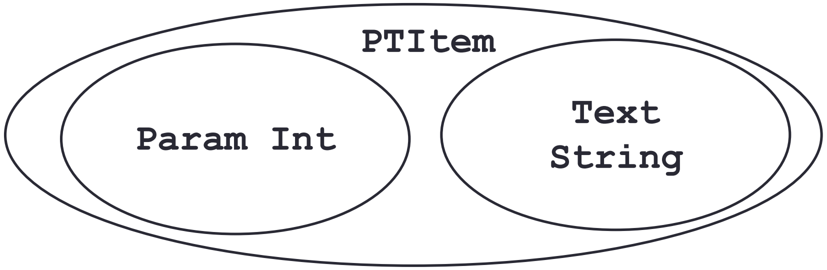
- Type OpenMode can be created by three data constructors.

- Type **`OpenMode`** has three values:
  - **`ReadOnly`**
  - **`WriteOnly`**
  - **`ReadWrite`**

- **`Bool`** is an enumeration type.

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

Union Type



- Similar to **`union`** in C programming language

```
data PTItem = Param Int | Text String
```

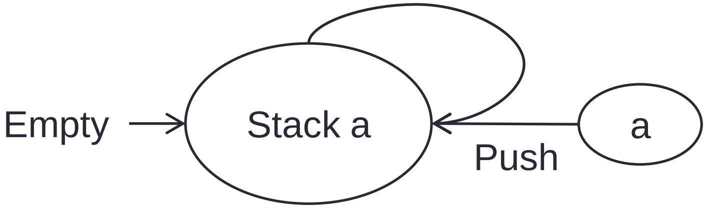
- Values of **`PTItem`** are either **`Param`** with integer or **`Text`** with String.

```
Text "daikon"
Param 5

isText::PTItem -> Bool
isText (Text _) = True
isText (Param _) = False

text::PTItem -> String
text (Text s) = s
text (Param _) = "(param)"
```

Recursive Type



- Type declaration may refer itself.

```
data Stack a = Empty | Push a (Stack a)
```

- Creating values of **`Stack a`**

```
Emtpy
Push 1 Empty
Push 2 (Push 1 Empty)
Push 3 (Push 2 (Push 1 Empty))
```

- Accessing values of **`Stack a`**

```
isEmpty::Stack a -> Bool
isEmpty Empty = True
isEmpty (Push _ _) = False

top::Stack a -> a
top (Push x _) = x

pop::Stack a -> Stack a
pop (Push _ s) = s
```

type Declaration

type $T\ v_1\ v_2\ \ldots = t$

type constructor

type variables

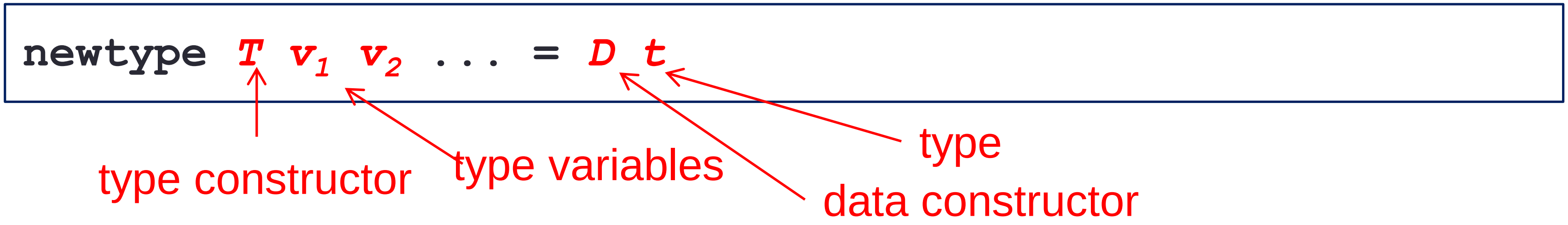
type

- Creating a type by renaming an existing type
  - No data constructor

```
type MyList a = [a]
```

- **`MyList a`** is just an alias of **`[a]`**.
  - Any functions for **`[a]`** can be used for **`MyList a`**.

newtype Declaration



newtype $T\ v_1\ v_2\ \ldots = D\ t$

type constructor

type variables

type

data constructor

- Creating a type by using an existing type
  - Has data constructor

```
newtype StackNT a = MKStackNT [a]

data StackNT a = MKStackNT [a]
```

- Similar to `data` declaration with only one data constructor
  - Representation inside Haskell is simpler for newtype.
  - `StackNT a` is represented as just `[a]`.

Type Class

- Restriction to polymorphism
  - `sort :: [a] -> [a]`
  - `sort` cannot sort arbitrary list, but only with order relation.

- Type class (or just class)
  - set of types
  - A type in a class needs to implement certain class methods.

- Example: `Ord` class
  - Values of a `Ord` class type can be compared.

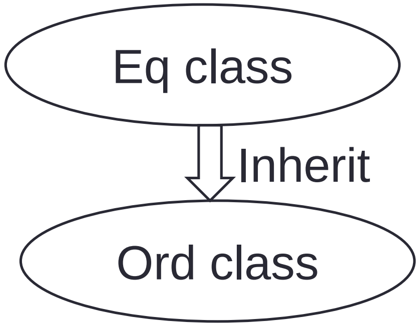
```
sort::(Ord a) => [a] -> [a]
```

- `(Ord a) =>` specifies the restriction to type variable `a`.
  - `a` needs to be a type on Ord class.

Inheritance

- Classes may have inheritance relation.

- Example: Eq class
  - Eq class is a super class of Ord class
  - Eq class has (==) as a class method



super class
class methods: **`(==)`** , ...

sub class
class methods: **`(<=)`** , **`(>=)`** , ...

Class Declaration

- **`Eq`** class

```
class Eq a where
  (==), (/=) :: a -> a -> Bool    -- declaration of class methods

  x == y = not (x /= y)           -- default implementation of (==)
  x /= y = not (x == y)           -- default implementation of (/=)
```

- **`Ord`** class (**`Eq`** as super class)

```
class (Eq a) => (Ord a) where
  compare              :: a -> a -> Ordering
  (<), (<=), (>), (>=) :: a -> a -> Bool
  min, max             :: a -> a -> a

  compare x y | x == y    = EQ
              | x <= y    = LT
              | otherwise = GT
  x <= y  = compare x y /= GT
  x <  y  = compare x y == LT
  x >= y  = compare x y /= LT
  x >  y  = compare x y == GT
  max x y | x <= y    =  y
          | otherwise =  x
  min x y | x <= y    =  x
          | otherwise =  y
```

instance Declaration

- Declaring a type is an instance of a class

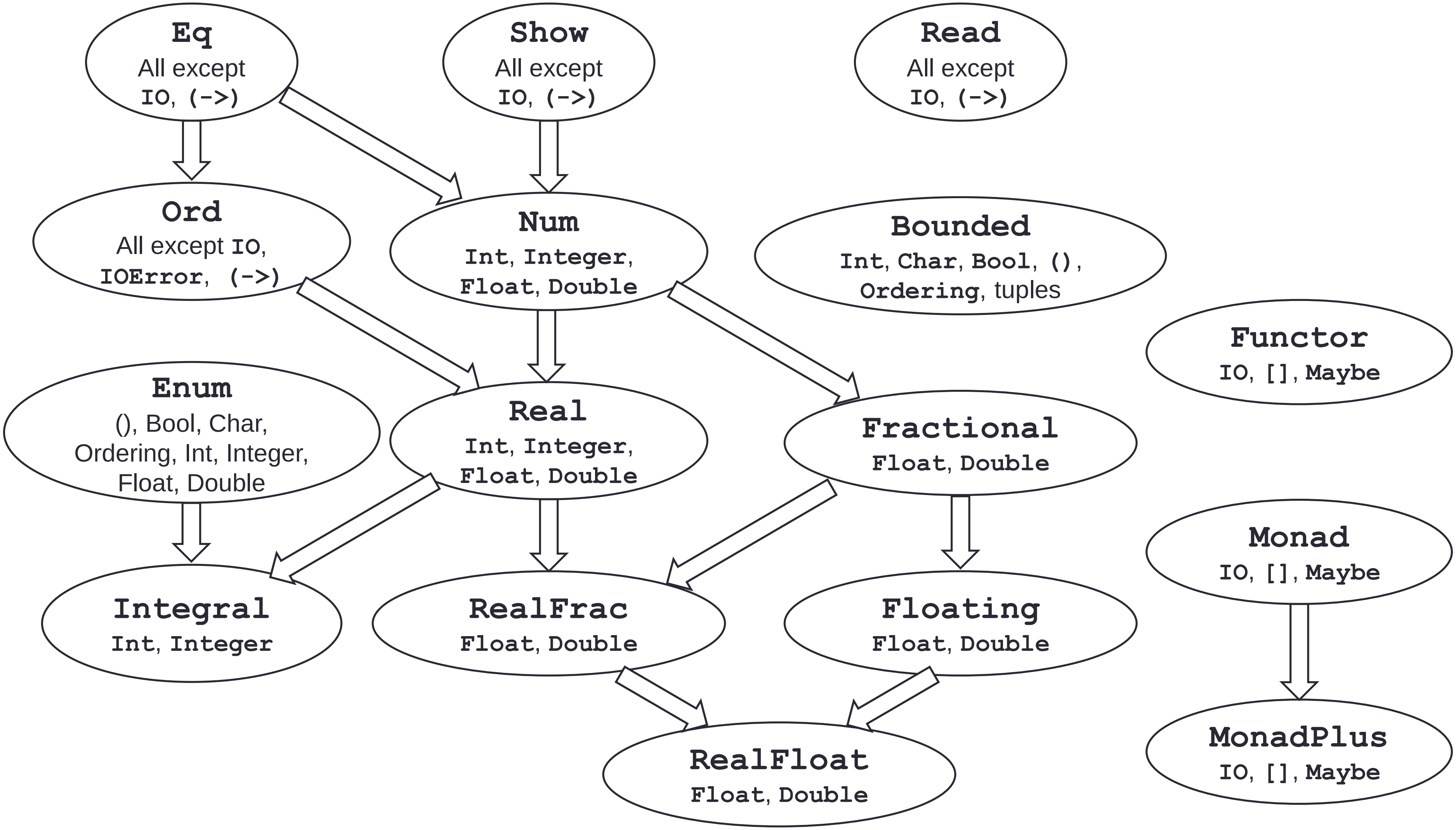
```
data Anchor = A String String

instance Eq Anchor where
  (A u l) == (A u' l') = (u == u') && (l == l')
```

- **`deriving`** declaration
  - If the implementation is natural and clear, let system implements them.
  - Available for: **`Eq`**, **`Ord`**, **`Enum`**, **`Bounded`**, **`Show`**, **`Read`**

```
data Anchor = A String String  deriving (Eq, Show)
```

Important Classes



`Eq`
All except `IO`, `(->)`

`Show`
All except `IO`, `(->)`

`Read`
All except `IO`, `(->)`

`Ord`
All except `IO`, `IOError`, `(->)`

`Num`
`Int`, `Integer`, `Float`, `Double`

`Bounded`
`Int`, `Char`, `Bool`, `()`, `Ordering`, tuples

`Functor`
`IO`, `[]`, `Maybe`

`Enum`
(), Bool, Char, Ordering, Int, Integer, Float, Double

`Real`
`Int`, `Integer`, `Float`, `Double`

`Fractional`
`Float`, `Double`

`Monad`
`IO`, `[]`, `Maybe`

`Integral`
`Int`, `Integer`

`RealFrac`
`Float`, `Double`

`Floating`
`Float`, `Double`

`MonadPlus`
`IO`, `[]`, `Maybe`

`RealFloat`
`Float`, `Double`

Example: Rational Number

- A rational number consists of two numbers: numerator and denominator
  - Declare the data type as a pair of integers

```
data Rat = Rat Integer Integer

main = print $ Rat 2 3
```

- Cannot **`print`** by default.
  - **`print`** is only available when **`show`** method is defined.
  - **`print::Show a => a -> IO ()`**

```
data Rat = Rat Integer Integer deriving Show

main = print $ Rat 2 3
```

- This works, but it shows `Rat 2 3' as it is.
- Define show method:

```
data Rat = Rat Integer Integer

instance Show Rat where
  show (Rat x y) = show x ++ "/" ++ show y

main = print $ Rat 2 3
```

Rational Number (cont.)

- Addition and multiplication of rational numbers?
- Define arithmetic functions
  - Use data constructor pattern

```
data Rat = Rat Integer Integer

instance Show Rat where
  show (Rat x y) = show x ++ "/" ++ show y

add::Rat -> Rat -> Rat
add (Rat x y) (Rat u v) = Rat (x * u + y * v) (y * v)

main = print $ add (Rat 1 2) (Rat 1 6)
```

Rational Number (cont.)

- Would like to use `(+)` and `(*)`
- Make it an instance of **`Num`** class.

```
class Num a where
   (+) :: a -> a -> a
   (*) :: a -> a -> a
  negate :: a -> a               -- or (-):: a -> a -> a
  abs :: a -> a
  signum :: a -> a               -- abs x * signum x == x needs to hold
  fromInteger :: Integer -> a
```

- Implement above 6 methods to make **`Rat`** an instance of **`Num`** class.

```
instance Num Rat where
   (Rat x y) + (Rat u v) = Rat (x * u + y * v) (y * v)
   (Rat x y) * (Rat u v) = Rat (x * u) (y * v)
  negate (Rat x y) = Rat (- x) y
  abs (Rat x y) = Rat (abs x) (abs y)
  signum (Rat x y) | x == 0     = fromInteger 0
                   | x * y > 0 = fromInteger 1
                   | otherwise = fromInteger (-1)
  fromInteger x = Rat x 1
```

```
main = print $ Rat 1 2 + Rat 1 6
```

Exercise 9-1

rat.hs

```haskell
import System.Environment

data Rat = Rat Integer Integer

instance Show Rat where
  show (Rat x y) = show x ++ "/" ++ show y

instance Num Rat where
  ...

main = do args <- getArgs
          let x = read (args !! 0)
          let y = read (args !! 1)
          let u = read (args !! 2)
          let v = read *args !! 3)
          print $ Rat x y + Rat u v
          print $ Rat x y - Rat u v
          print $ Rat x y * Rat u v
```

```
% ./rat 2 3 4 5
22/15
-2/15
8/15
5/6
% ./rat 1 3 2 3
1
-1/3
2/9
1/2
%
```

- Complete the implementation of rational number:
  - Reduce fractions to irreducible ones: $\frac{3}{6} \rightarrow \frac{1}{2}$
  - Make denominators always positive: $\frac{2}{-3} \rightarrow \frac{-2}{3}$
  - If the result is integer, print is as integer: $\frac{2}{1} \rightarrow 2$
  - Make is an instance of **`Fractional`**.