

CSCE 314

Programming Languages

Haskell: Types, Currying and Polymorphism

Dr. Hyunyoung Lee

Types

A type is a collection of related values. For example,

- **Bool** contains the two logical values **True** and **False**
- **Int** contains values $-2^{63}, \dots, -1, 0, 1, \dots, 2^{63} - 1$

Data
constructed

If evaluating an expression e would produce a value of type T , then e has type T , written

$e :: T$

if e makes type T , then e is type T

Every well-formed expression has a type, which can be automatically calculated at *compile time* using a process called type inference

→ evaluate at compile

Type Errors

Applying a function to one or more arguments of the wrong type is called a type error

```
> 1 + False  
Error
```

1 is a number and False is a logical value, but + requires two numbers

Static type checking – all type errors are found at compile time, which makes programs safer and faster by removing the need for type checks at run time

$(1::\text{Int}) + (2.3::\text{Float}) \rightarrow \text{type error}$
but $1 + 2.3$ will have no error

Type Annotations

Programmer can (and at times must) annotate expressions with type in the form $e :: T$

For example,

e has type T

- `True :: Bool`
- `5 :: Int` `-- type is really (Num t) => t`
- `(5 + 5) :: Int` `-- likewise`
- `(7 < 8) :: Bool`

Some expressions can have many types, e.g.,

`5 :: Int, 5 :: Integer, 5 :: Float`

GHCi command `:type e` shows the type of (the result of) `e`

```
> not False
True
```

```
> :type not False
not False :: Bool
```

Basic Types

Haskell has a number of basic types, including:

`Bool`

- logical values

`Char`

- single characters

`String`

- lists of characters `type String = [Char]`

`Int`

- fixed-precision integers

`Integer`

- arbitrary-precision integers

`Float`

- single-precision floating-point numbers

`Double`

- double-precision floating-point numbers

List Types

List \rightarrow for same type

A list is sequence of values of the same type:

```
[False, True, False] :: [Bool]
['a', 'b', 'c']      :: [Char]
"abc"                :: [Char]
[[True, True], []]   :: [[Bool]]
```

*↑ inferred
to be a list
of Booleans*

Note:

- `[t]` has the type list with elements of type `t`
- The type of a list says nothing about its length
- The type of the elements is unrestricted
- Lists can be infinite: `l = [1..]`

Tuple Types

Tuple → different types
Must specify how many components → can't be changed

A tuple is a sequence of values of different types:

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

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

```
("Howdy", (True, 2)) :: ([Char], (Bool, Int))
```

Note:

- (t_1, t_2, \dots, t_n) is the type of n -tuples whose i -th component has type t_i for any i in $1 \dots n$
- The type of a tuple encodes its size
- The type of the components is unrestricted
- Tuples with *arity* one are not supported: (t) is parsed as t , parentheses are ignored

Function Types

A function is a mapping from values of one type (T1) to values of another type (T2), with the type

T1 → T2

↑
Type constructor
arrow

$f :: a \rightarrow b$

```
not      :: Bool -> Bool
isDigit  :: Char -> Bool
toUpper  :: Char -> Char
(&&)     :: Bool -> Bool -> Bool
```

$f :: a \rightarrow b$
↑
function
↑
takes in type A
↑
Returns type B

Note:

The argument and result types are unrestricted. Functions with multiple arguments or results are possible using lists or tuples:

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

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

One parameter functions!

Curried Functions

*Curried function,
a function with more
than one argument*

Functions with multiple arguments are also possible by returning functions as results:

```
add :: (Int,Int) → Int
add (x,y) = x+y

add' :: Int → (Int → Int)
add' x y = x+y
```

add' takes an int x and **returns a function add' x**. In turn, this function takes an int y and returns the result x+y

Note:

- add and add' produce the same final result, but add takes its two arguments at the same time, whereas add' takes them one at a time
- Functions that take their arguments one at a time are called **curried** functions, celebrating the work of Haskell Curry on such functions

Functions with more than two arguments can be carried by returning nested functions:

```
mult      :: Int → (Int → (Int → Int))  
mult x y z = x*y*z
```

mult takes an integer x and returns a function mult x , which in turn takes an integer y and returns a function mult x y , which finally takes an integer z and returns the result $x*y*z$

Note:

- Functions returning functions: an example of higher-order functions
- Unless tupling is explicitly required, all functions in Haskell are normally defined in curried form

Why is Currying Useful?

Curried functions are more flexible than functions on tuples, because useful functions can often be made by partially applying a curried function

For example:

```
add' 1 :: Int -> Int
take 5 :: [a] -> [a]
drop 5 :: [a] -> [a]
```

```
map :: (a->b) -> [a] -> [b]
map f []      = []
map f (x:xs) = f x : map f xs
```

`> map (add' 1) [1,2,3]` $= (f\ 1) : (f\ 2) : (f\ 3)$
`[2,3,4]` $\underbrace{\hspace{1cm}}_f \quad \underbrace{[2,3]}_{[3]} \quad 2:3:4:[]$
 $= [2,3,4]$

Currying Conventions

To avoid excess parentheses when using curried functions, two simple conventions are adopted:

1. The arrow \rightarrow (type constructor) associates to the right

`Int \rightarrow Int \rightarrow Int \rightarrow Int`

Means `Int \rightarrow (Int \rightarrow (Int \rightarrow Int))`

2. As a consequence, it is then natural for function application to associate to the left

`mult x y z`

Means `((mult x) y) z`

Polymorphic Functions

Function that
→ can use diff
types of arguments

A function is called polymorphic (“of many forms”) if its type contains one or more type variables
Thus, polymorphic functions work with many types of arguments

$\text{length} :: [a] \rightarrow \text{Int}$

for any type a , length takes a list of values of type a and returns an integer

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

for any type a , id maps a value of type a to itself

$\text{head} :: [a] \rightarrow a$

$\text{take} :: \text{Int} \rightarrow [a] \rightarrow [a]$

a is a type variable

Polymorphic Types

Type variables can be instantiated to different types in different circumstances:

```
> length [False,True]
```

```
2
```

```
> length [1,2,3,4::Int]
```

```
4
```

a = Bool

a = Int

expression	polymorphic type	type variable bindings	resulting type
id	$a \rightarrow a$	a=Int	Int \rightarrow Int
id	$a \rightarrow a$	a=Bool	Bool \rightarrow Bool
length	$[a] \rightarrow \mathbf{Int}$	a=Char	[Char] \rightarrow Int
fst	$(a, b) \rightarrow a$	a=Char, b=Bool	Char
snd	$(a, b) \rightarrow b$	a=Char, b=Bool	Bool
([], [])	$([a], [b])$	a=Char, b=Bool	([Char], [Bool])

Type variables must begin with a lower-case letter, and are usually named a, b, c, etc.

Overloaded Functions

A polymorphic function is called overloaded if its type contains one or more class constraints

`sum :: Num a => [a] -> a`

for any **numeric** type `a`,
sum takes a list of values
of type `a` and returns a
value of type `a`

Constrained type variables can be instantiated to
any types that satisfy the constraints:

`> sum [1,2,3]`

`6`

`a = Int`

`> sum [1.1,2.2,3.3]`

`6.6`

`a = Float`

`> sum ['a','b','c']`

`ERROR`

Char is not a numeric type

Class Constraints

Recall that polymorphic types can be instantiated with all types, e.g.,

`id :: t -> t` `length :: [t] -> Int`

This is when no operation is subjected to values of type `t`

What are the types of these functions?

`min :: Ord a => a -> a -> a` *(x, y, return value)*

`min x y = if x < y then x else y`

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

`elem x (y:ys) | x == y = True`

`elem x (y:ys) = elem x ys`

`elem x [] = False`

Type variables can only be bound to types that satisfy the constraints

`Ord a` and `Eq a` are class constraints

Type Classes

Constraints arise because values of the generic types are subjected to operations that are not defined for all types:

```
min :: Ord a => a -> a -> a
min x y = if x < y then x else y

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

Ord and Eq are **type classes**:

Num (Numeric types)

Eq (Equality types)

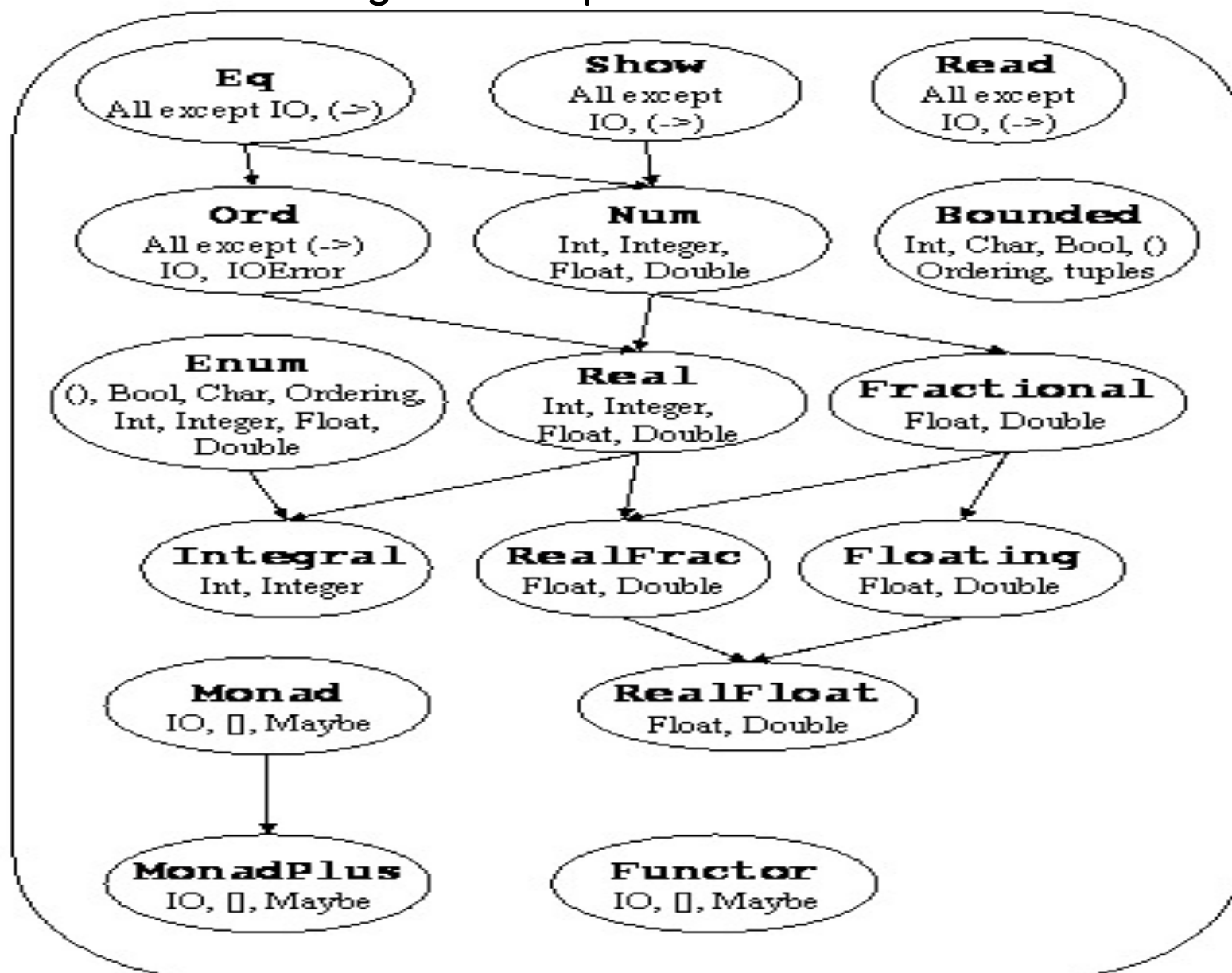
Ord (Ordered types)

(+)	:: Num a => a -> a -> a
(==)	:: Eq a => a -> a -> Bool
(<)	:: Ord a => a -> a -> Bool

Haskell 98 Class Hierarchy

For detailed explanation, refer

<http://www.haskell.org/onlinereport/basic.html>



The Eq and Ord Classes

class Eq a where

(==), (/=) :: a -> a -> Bool

x /= y = not (x == y)

x == y = not (x /= y)

class (Eq a) => Ord a where

compare :: a -> a -> Ordering

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

max, min :: 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

The Enum Class

```
class Enum a where
  toEnum    :: Int -> a
  fromEnum  :: a -> Int
  succ, pred :: a -> a
  . . .
```

-- Minimal complete definition: toEnum, fromEnum

Note: these methods only make sense for types that map injectively into Int using fromEnum and toEnum

```
succ = toEnum . (+1) . fromEnum
```

```
pred = toEnum . (subtract 1) . fromEnum
```

The Show and Read Classes

class Show a where
 show :: a -> String

class Read a where
 read :: String -> a

Many types are showable and/or readable

> show 10

"10"

> show [1,2,3]

"[1,2,3]"

> read "10" :: Int

10

> read "[1,2,3]" :: [Int]

[1,2,3]

> map (* 2.0) (read "[1,2]")

[2.0,4.0]

Hints and Tips

When defining a new function in Haskell, it is useful to begin by writing down its type

Within a script, it is good practice to state the type of every new function defined

When stating the types of polymorphic functions that use numbers, equality or orderings, take care to include the necessary class constraints

Exercises

(1) What are the types of the following values?

```
['a', 'b', 'c']
```

```
('a', 'b', 'c')
```

```
[(False, '0'), (True, '1')]
```

```
([False, True], ['0', '1'])
```

```
[tail, init, reverse]
```

(2) What are the types of the following functions?

`second xs = head (tail xs)`

`swap (x,y) = (y,x)`

`pair x y = (x,y)`

`double x = x*2`

`palindrome xs = reverse xs == xs`

`twice f x = f (f x)`

`lessThanHalf x y = x * 2 < y`

(3) Check your answers using GHCi.