## Types and Typeclasses

Yan Huang

#### What is a Type?

# A <u>type</u> is a name for a collection of related values. For example, in Haskell the basic type

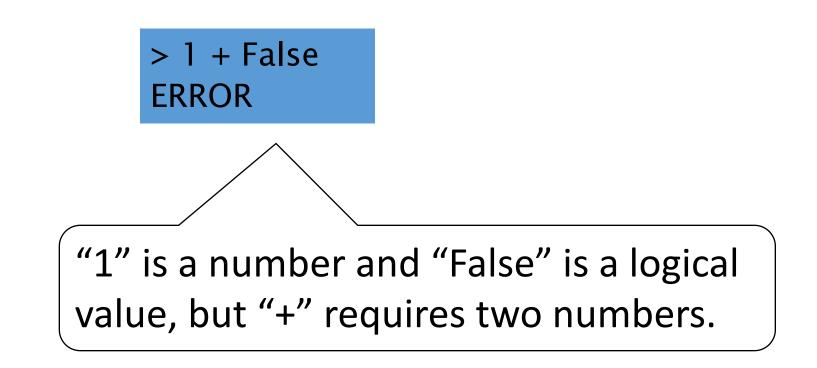
Bool

contains the two logical values:



#### Type Errors

# Applying a function to arguments of mismatching types results a <u>type error</u>.



## Types in Haskell

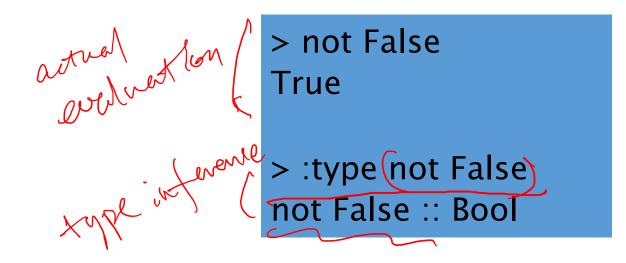
• If evaluating an expression e would produce a value of type t, then e <u>has type</u> t, written



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

All type errors are found at compile time, which makes programs <u>safer and faster</u> by removing the need for type checks at run time.

In GHCi, the <u>:type</u> command calculates the type of an expression, without evaluating it:



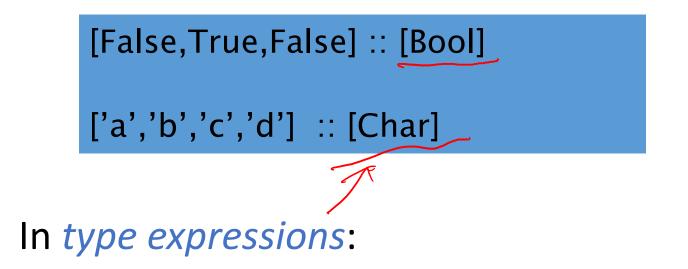
### Basic Types

#### Haskell has a number of <u>basic types</u>, including:

	Bool	- logical values
	Char	- single characters
	String	- strings of characters - strings of characters
	Int	- fixed-precision integers
$\smile$	Integer	- arbitrary-precision integers
	Float	- floating-point numbers

#### List Types

A <u>list</u> is sequence of values of the <u>same</u> type:

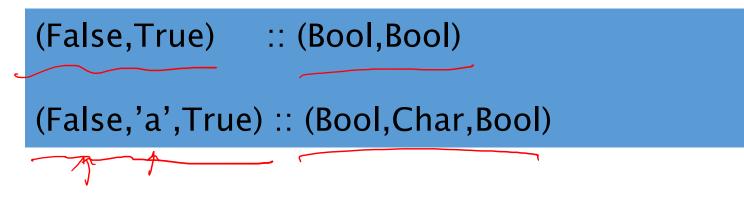


[t] is the type of lists with elements of type t. type variable • The type of a list says nothing about its length:

```
[False,True] :: [Bool]
[False,True,False] :: [Bool]
```

• The type of the elements is unrestricted. For example, we can have lists of lists:

#### Tuple Types A <u>tuple</u> is a sequence of values of potentially <u>different</u> types:



In *type expressions*:

 $(t_1, t_2, ..., t_n)$  is the type of n-tuples whose *i*-th components have type  $t_i$  for any *i* in 1,...,*n*.

Note:  
(Int, Boil)  
• The type of a tuple encodes its size:  

$$f(Bool, Int)$$
  
(False, True) :: (Bool, Bool)  
(False, True, False) :: (Bool, Bool, Bool)  
 $f(M, M, M)$ 

• The type of the components is unrestricted:

Function Types

A <u>function</u> is a mapping from values of one type to values of another type:

not :: Bool  $\leftrightarrow$  Bool even :: Int  $\rightarrow$  Bool

In type expressions:

 $t1 \rightarrow t2$  is the type of functions that map values of type t1 to values to type t2.

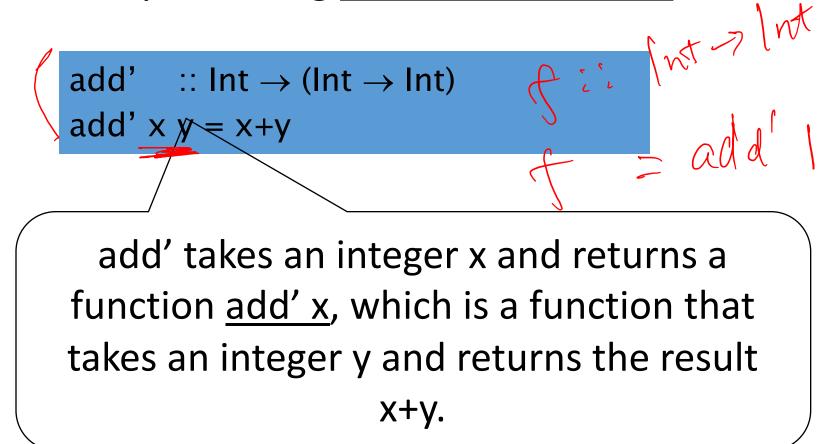
Arrow  $\rightarrow$  is typed as "->" in editors.

The argument and result types are unrestricted. For example, functions with multiple arguments or results are possible using lists or tuples:

> add ::  $(Int,Int) \rightarrow Int$ add (x,y) = x+yzeroto ::  $Int \rightarrow [Int]$ zeroto n = [0..n]

#### **Curried Functions**

Functions with multiple arguments are also possible by returning <u>functions as results</u>:



 add and add' produce the same final result, but add takes its two arguments at the same time in a tuple, whereas add' takes them one at a time:

```
add :: (Int,Int) \rightarrow Int
add' :: Int \rightarrow (Int \rightarrow Int)
```

 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 curried by returning nested functions:

 $\begin{array}{ll} \text{mult} & :: \text{Int} \rightarrow (\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})) \\ \text{mult x y } z = x^*y^*z \end{array}$ 

mult takes an integer x and returns a function <u>mult x</u>, which in turn takes an integer y and returns a function <u>mult x y</u>, which finally takes an integer z and returns the result x\*y\*z.

## Why is Currying Useful?

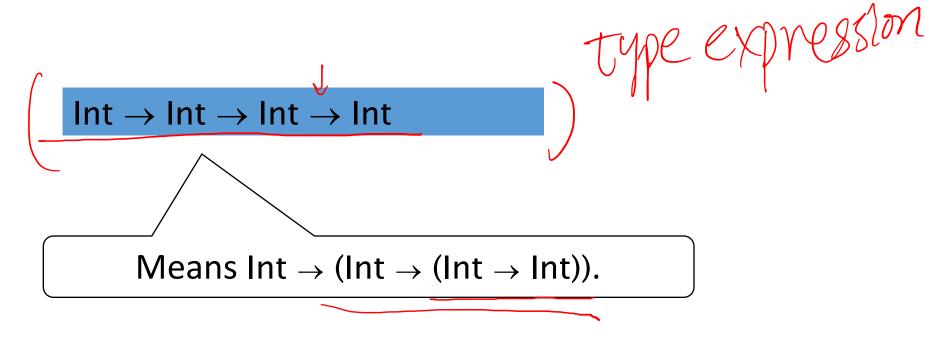
Curried functions are more flexible than functions on tuples, because useful functions can often be made by <u>partially applying</u> a curried function.

add' 1 :: Int  $\rightarrow$  Int take 5 :: [Int]  $\rightarrow$  [Int] drop 5 :: [Int]  $\rightarrow$  [Int]

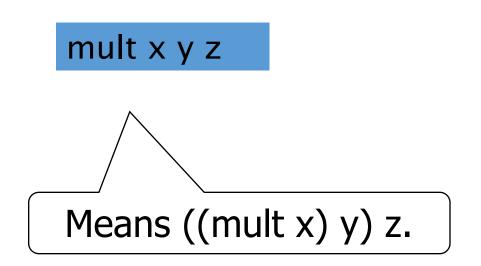
#### **Currying Conventions**

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

• The arrow  $\rightarrow$  in type expressions associates to the <u>right</u>.

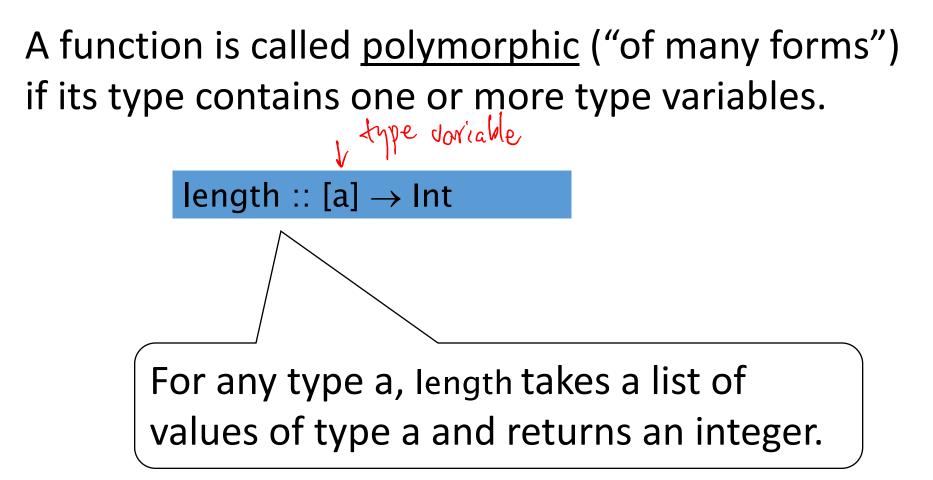


#### But function application associates to the left.



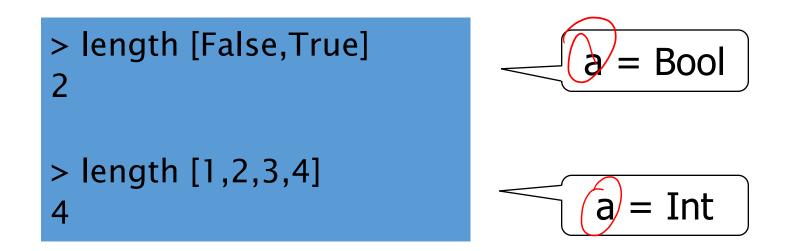
Unless tupling is explicitly required, all functions in Haskell are normally defined in curried form.

#### **Polymorphic Functions**



#### Type Variables

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



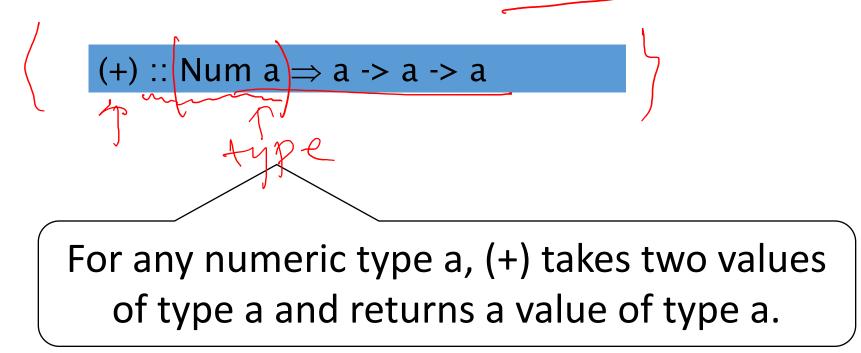
• Type variables must begin with a lower-case letter, and are usually named a, b, c, etc.

Many of the functions defined in the standard prelude are polymorphic. For example:

fst :: (a,b)  $\rightarrow$  a fst (1, 'c') => head [5,6,1,2] => 5 head ::  $[a] \rightarrow a$ take :: Int  $\rightarrow$  [a]  $\rightarrow$  [a] +ake 5 [1...10] => [123, 4,5]  $z' \rho$  $zip :: [a] \rightarrow [b] \rightarrow [(a,b)]$ id ::  $a \rightarrow a$ 

#### **Overloaded Functions**

A polymorphic function is called <u>overloaded</u> if its type contains one or more class constraints.



## Type Constraints

Haskell has a number of type classes, including:

- Num | Eq - | Ord -
- Numeric types  $\{t_1, t_2, t_3, \dots, t_n\}$

Int

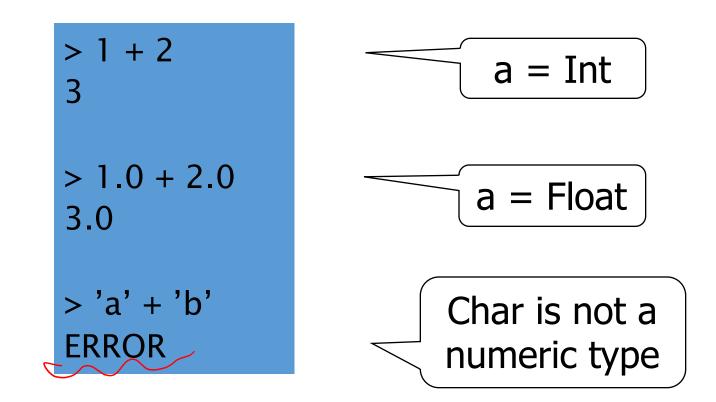
- Equality types
- Ordered types

For example:

(+) :: Num  $a \Rightarrow a \rightarrow a \rightarrow a$ (==) :: Eq  $a \Rightarrow a \rightarrow a \rightarrow Bool$ (<) :: Ord  $a \Rightarrow a \rightarrow a \rightarrow Bool$ 

#### Type Constraints

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



#### Typeclass Example

#### class Num a where

 $\sim$ 

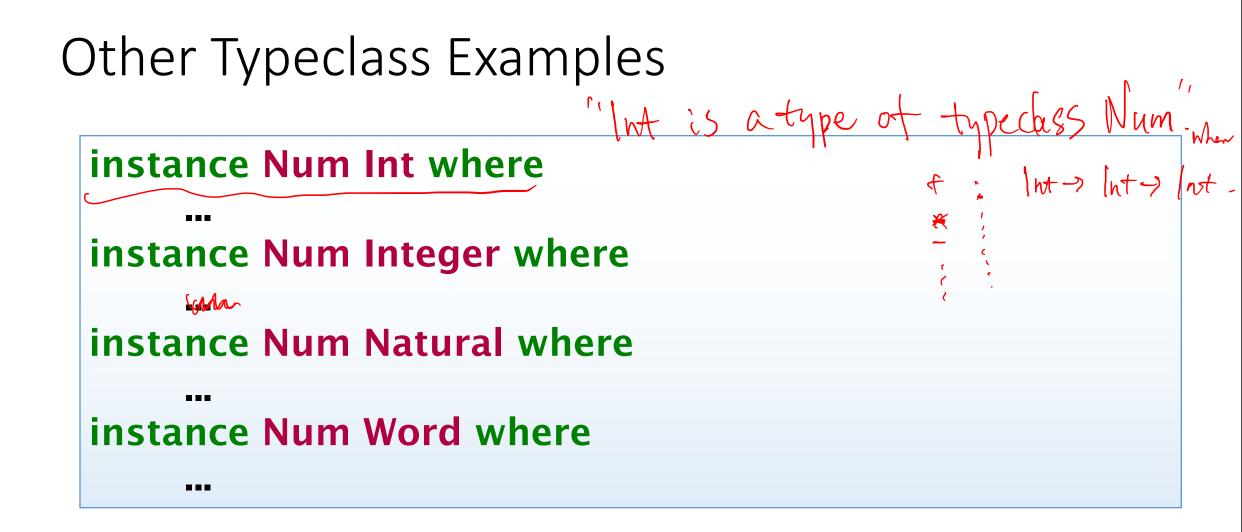
(+), (-), (*) negate ~	:: <u>a -&gt; a -&gt; a</u>
negate ∽	:: a -> a
abs	:: a -> a
signum	:: a -> a
fromInteger	:: Integer ->

a

~ x - y	= x + negate y
negate x	= 0 - x

#### Typeclass Example

```
instance Num Int where
  x + y = .....
  x - y = .....
  negate x = ....
  x * y = ....
  abs n = ...
  signum n = ...
     fromInteger i = ...
```



## Haskell's Automatic Type Inference

double X = RFX

• How does your compiler automatically infer their types?





## Haskell's Automatic Type Inference chass Num a abere • How does your compiler automatically infer their types? pes: 外に、 ん-> ん-> の times x y = x \* yAlghedric body times :: (Num 4) => A-> A-> A of the of function the function the

# • How does your compiler automatically infer their types? (Num a) = factorial n = product [1 n]

factorial :: (Mun 0)=>a-> Q

•

#### 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)

(3) Check your answers using GHCi.