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:

False   True

Type Errors

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

```
> 1 + False
ERROR
```

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

Types in Haskell

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

`e :: t`

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:

actual evaluation

```
> not False
True

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

type inference

Basic Types

Haskell has a number of basic types, including:

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

:type "abc"

"abc" :: [Char]

List Types

A list is sequence of values of the same type:

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

['a','b','c','d']  :: [Char]
```

In *type expressions*:

[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:

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

Tuple Types

A tuple is a sequence of values of potentially different types:

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

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

In *type expressions*:

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

Note:

(Int, Bool)

≠ (Bool, Int)

- The type of a tuple encodes its size:

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

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

(Int, Int, Int)

≠ (Int, Int)

- The type of the components is unrestricted:

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

(True,['a','b'])  :: (Bool,[Char])
```

Function Types

A function is a mapping from values of one type to values of another type:

```
not  :: Bool → Bool

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

zeroto     :: Int → [Int]
zeroto n   = [0..n]
```

Curried Functions

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

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

f :: Int → Int

f = add' 1

add' takes an integer x and returns a function add' x, which is a function that takes an integer y and returns the result x+y.

- 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) → Int

add’ :: Int → (Int → 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:

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

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 → Int

take 5 :: [Int] → [Int]

drop 5 :: [Int] → [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 right.

type expression

Int $\rightarrow$ Int $\rightarrow$ Int $\rightarrow$ Int

Means Int $\rightarrow$ (Int $\rightarrow$ (Int $\rightarrow$ Int)).

But function application associates to the <u>left</u>.

```
mult x y z
```

Means ((mult x) y) z.

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

Polymorphic Functions

A function is called polymorphic ("of many forms") if its type contains one or more type variables.

type variable

```
length :: [a] → Int
```

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

Type Variables

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

```
> length [False,True]
2

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

a = Bool

a = Int

- 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) → a
head :: [a] → a
take :: Int → [a] → [a]
zip  :: [a] → [b] → [(a,b)]
id   :: a → a
```

fst (1, 'c') => 1

head [5, 6, 1, 2] => 5

take 5 [1 .. 10] => [1,2,3, 4,5]

zip

Overloaded Functions

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

```
(+) :: Num a ⇒ a -> a -> a
```

type

For any numeric type a, (+) takes two values of type a and returns a value of type a.

Type Constraints

Haskell has a number of type classes, including:

Int

$\{t_1, t_2, t_3 \ldots t_n\}$

Num	-	Numeric types
Eq	-	Equality types
Ord	-	Ordered types

For example:

$(+) \;::\; Num\ a \Rightarrow a \to a \to a$

$(==) \;::\; Eq\ a \Rightarrow a \to a \to Bool$

$(<) \;::\; Ord\ a \Rightarrow a \to a \to Bool$

Type Constraints

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

```
> 1 + 2
3

> 1.0 + 2.0
3.0

> 'a' + 'b'
ERROR
```

a = Int

a = Float

Char is not a numeric type

Typeclass Example

```
class Num a  where
  (+), (-), (*)     :: a -> a -> a
  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 = ...
```

Other Typeclass Examples

"Int is a type of typeclass Num" where

f : Int -> Int -> Int

```
instance Num Int where
        ...
instance Num Integer where
        ...
instance Num Natural where
        ...
instance Num Word where
        ...
```

Haskell's Automatic Type Inference

double x = x+x

- How does your compiler automatically infer their types?

```
first x y = x
```

first :: a -> b -> a

```
f x = x
```

f :: a -> a

f x = x+1

f :: (Num a) => a -> a

Haskell's Automatic Type Inference

class Num a where

- How does your compiler automatically infer their types?

(*) :: a -> a -> a

```
times x y = x * y
```

signature of the function

body of the function

times :: (Num a) => a -> a -> a

Haskell's Automatic Type Inference

- How does your compiler automatically infer their types?

```
factorial n = product [1..n]
```

product :: (Num a) => [a] -> a

factorial :: (Num a) => a -> a

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.