Types and Typeclasses

Yan Huang
What is a Type?

A type 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 type error.

> 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 **safer and faster** by removing the need for type checks at run time.

In GHCi, the `:type` command calculates the type of an expression, without evaluating it:

```ghci
c> 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** - strings of characters
- **Int** - fixed-precision integers
- **Integer** - arbitrary-precision integers
- **Float** - floating-point numbers
List Types

A list is sequence of values of the same type:

\[[\text{False, True, False}] :: [\text{Bool}]\]
\(['a', 'b', 'c', 'd'] :: [\text{Char}]\)

In type expressions:

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

\[
\begin{align*}
\text{[False,True]} &\ :: \ \text{[Bool]} \\
\text{[False,True,False]} &\ :: \ \text{[Bool]}
\end{align*}
\]

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

\[
\begin{align*}
\text{[[’a’],[’b’,’c’]]} &\ :: \ \text{[[Char]]}
\end{align*}
\]
Tuple Types

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

\[(\text{False}, \text{True}) :: (\text{Bool}, \text{Bool})\]

\[(\text{False}, 'a', \text{True}) :: (\text{Bool}, \text{Char}, \text{Bool})\]

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:

• The type of a tuple encodes its size:

\[(\text{False, True}) :: \text{(Bool, Bool)}\]
\[(\text{False, True, False}) :: \text{(Bool, Bool, Bool)}\]

• The type of the components is unrestricted:

\[('a', (\text{False, 'b'})) :: \text{(Char, (Bool, Char))}\]
\[(\text{True, ['a', 'b']}) :: \text{(Bool, [Char])}\]
Function Types

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

\[
\text{not :: Bool } \rightarrow \text{ Bool}
\]

\[
\text{even :: Int } \rightarrow \text{ Bool}
\]

In type expressions:

\[
t1 \rightarrow t2 \text{ is the type of functions that map values of type } t1 \text{ to values to type } t2.
\]
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:

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

zeroto :: Int -> [Int]
zeroto n = [0..n]
```
Functions with multiple arguments are also possible by returning functions as results:

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

`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:

\[
\begin{align*}
\text{add} &\colon (\text{Int},\text{Int}) \to \text{Int} \\
\text{add'} &\colon \text{Int} \to (\text{Int} \to \text{Int})
\end{align*}
\]

• Functions that take their arguments one at a time are called \textit{curried} functions, celebrating the work of Haskell Curry on such functions.
Functions with more than two arguments can be curried by returning nested functions:

\[
\text{mult} :: \text{Int} \rightarrow (\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})) \\
\text{mult} x y z = x \times y \times z
\]

\(\text{mult}\) takes an integer \(x\) and returns a function \(\text{mult} x\), which in turn takes an integer \(y\) and returns a function \(\text{mult} x y\), which finally takes an integer \(z\) and returns the result \(x \times y \times z\).
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.

```
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 $\to$ in type expressions associates to the right.

\[
\text{Int} \to \text{Int} \to \text{Int} \to \text{Int}
\]

Means \[
\text{Int} \to (\text{Int} \to (\text{Int} \to \text{Int})).
\]
But function application associates to the left.

\[
\text{mult } x \ y \ z
\]

Means \(((\text{mult } x) \ y) \ z\).

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.

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

For any type \( a \), \( \text{length} \) 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
  ```

- 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:

\[
\begin{align*}
\text{fst} & :: (a, b) \to a \\
\text{head} & :: [a] \to a \\
\text{take} & :: \text{Int} \to [a] \to [a] \\
\text{zip} & :: [a] \to [b] \to [(a, b)] \\
\text{id} & :: a \to a
\end{align*}
\]

\[
\begin{align*}
\text{fst} (1, 'c') & \Rightarrow 1 \\
\text{head} [5, 6, 1, 2] & \Rightarrow 5 \\
\text{take} 5 [1..10] & \Rightarrow [1, 2, 3, 4, 5] \\
\text{zip} & \Rightarrow \\
\end{align*}
\]
A polymorphic function is called **overloaded** if its type contains one or more class constraints.

\[ (+) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a \]

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:

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

For example:

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

- $a = \text{Int}$
- $a = \text{Float}$
- 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

instance Num Int where
    ...

instance Num Integer where

instance Num Natural where
    ...

instance Num Word where
    ...

"Int is a type of typeclass Num"
Haskell’s Automatic Type Inference

• How does your compiler automatically infer their types?

\[ \text{first } x \ y = x \]

\[ f \ x = x \]

\[ f \ x = x + 1 \]
Haskell’s Automatic Type Inference

• How does your compiler automatically infer their types?

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

```
class Num a where
  (*) :: a -> a -> a
```

```
times :: (Num a) => a -> a -> a
```
Haskell’s Automatic Type Inference

• How does your compiler automatically infer their types?

factorial n = product [1..n]

\[ \text{factorial} :: (\text{Num } a) \Rightarrow a \rightarrow a \]
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.