Types and Typeclasses

Stephen A. Edwards

Columbia University

Fall 2019
Haskell is **statically typed**: every expression’s type known at compile-time

Haskell has **type inference**: the compiler can deduce most types itself

Type names start with a **capital letter** (Int, Bool, Char, etc.)

GHci’s :t command reports the type of any expression

Read “::” as “is of type”
## Some Common Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Bool</code></td>
<td>Booleans: True or False</td>
</tr>
<tr>
<td><code>Char</code></td>
<td>A single Unicode character, about 25 bits</td>
</tr>
<tr>
<td><code>Int</code></td>
<td>Word-sized integers; the usual integer type. E.g., 64 bits on my x86_64 Linux desktop</td>
</tr>
<tr>
<td>Integer</td>
<td>Unbounded integers. Less efficient, so only use if you need really big integers</td>
</tr>
<tr>
<td><code>Float</code></td>
<td>Single-precision floating point</td>
</tr>
<tr>
<td><code>Double</code></td>
<td>Double-precision floating point</td>
</tr>
</tbody>
</table>
The Types of Functions

In a type, \(\rightarrow\) indicates a function

```haskell
Prelude> welcome x = "Hello " ++ x
Prelude> welcome "Stephen"
"Hello Stephen"
Prelude> :t welcome
welcome :: [Char] -> [Char]
```

“Welcome is a function that takes a list of characters and produces a list of characters”
Multi-argument functions are Curried

Haskell functions have exactly one argument. Functions with “multiple arguments” are actually functions that return functions that return functions.

Such “currying” is named after Haskell Brooks Curry, who is also known for the Curry-Howard Correspondence (“programs are proofs”).

Prelude> say x y = x++" to "++y
Prelude> :t say
say :: [Char] -> [Char] -> [Char]
Prelude> say "Hello" "Stephen"
"Hello to Stephen"

Prelude> :t say "Hello"
say "Hello" :: [Char] -> [Char]
Top-level Type Declarations

It is good style in .hs files to include type declarations for top-level functions

Best documentation ever: a precise, compiler-verified function summary

```haskell
-- addThree.hs
addThree :: Int -> Int -> Int -> Int
addThree x y z = x + y + z
```

```haskell
Prelude> :l addThree
[1 of 1] Compiling Main ( addThree.hs, interpreted )
Ok, one module loaded.
*Main> :t addThree
addThree :: Int -> Int -> Int -> Int
*Main> addThree 1 2 3
6
```
Polymorphism and Type Variables

Haskell has excellent support for polymorphic functions.

Haskell supports *parametric polymorphism*, where a value may be of *any* type.

Haskell also supports *ad hoc polymorphism*, where a value may be one of a *set of types* that support a particular group of operations.

**Parametric polymorphism: the `head` function**

```haskell
Prelude> :t head
head :: [a] -> a
```

Here, `a` is a *type variable* that ranges over *every possible type*.

**Ad hoc polymorphism: the `fst` function**

```haskell
Prelude> :t fst
fst :: (a, b) -> a
```

Here, `a` and `b` are distinct type variables, which may be *equal* or *different*. 
Ad Hoc Polymorphism and Type Classes

Haskell’s ad hoc polymorphism is provided by Type Classes, which specify a group of operations that can be performed on a type (think Java Interfaces)

```
Prelude> :t (==)
(==) :: Eq a => a -> a -> Bool
```

“The (==) function takes two arguments of type a, which must be of the Eq class, and returns a Bool”

Members of the Eq class can be compared for equality

A type may be in multiple classes; multiple types may implement a class
Common Typeclasses

**Eq**  
Equality: `==` and `/=`

**Ord**  
Ordered: Eq and `>`, `>=`, `<`, `<=`, `max`, `min`, and `compare`, which gives an Ordering: LT, EQ, or GT

**Enum**  
Enumerable: `succ`, `pred`, `fromEnum`, `toEnum` (conversion to/from Int), and list ranges

**Bounded**  
`minBound`, `maxBound`

**Num**  
Numeric: `+`, `(-)`, `(*)`, `negate`, `abs`, `signum`, and `fromInteger`

**Real**  
Num, Ord, and `toRational`

**Integral**  
Real, Enum, and `quot`, `rem`, `div`, `mod`, `toInteger`, `quotRem`, `divMod`

**Show**  
Can be turned into a string: `show`, `showList`, and `showsPrec` (operator precedence)

**Read**  
Opposite of Show: string can be turned into a value: `read` et al.
Ord, Enum, and Bounded Typeclasses

Prelude> :t (>)
(>) :: Ord a => a -> a -> Bool

Prelude> :t compare
compare :: Ord a => a -> a -> Ordering

Prelude> :t succ
succ :: Enum a => a -> a

Prelude> maxBound :: Int
9223372036854775807

Prelude> minBound :: Char
'\NUL'

Prelude> maxBound :: Char
'\1114111'

Prelude> minBound :: (Char, Char)
('\NUL','\NUL')
The Num Typeclass

Prelude> :t 42
42 :: Num p => p  -- Numeric literals are polymorphic

Prelude> :t (+)
(+) :: Num a => a -> a -> a  -- Arithmetic operators are, too

Prelude> :t 1 + 2
1 + 2 :: Num a => a

Prelude> :t (1 + 2) :: Int
(1 + 2) :: Int :: Int  -- Forcing the result type

Prelude> :t (1 :: Int) + 2
(1 :: Int) + 2 :: Int  -- Type of one argument forces the type

Prelude> :t (1 :: Int) + (2 :: Double)
<interactive>:1:15: error:
  * Couldn't match expected type 'Int' with actual type 'Double'
  * In the second argument of '(+)\', namely '(2 :: Double)'

In the expression: (1 :: Int) + (2 :: Double)
The Integral and Fractional Typeclasses

Prelude> :t div
div :: Integral a => a -> a -> a -- div is integer division
Prelude> :t toInteger
toInteger :: Integral a => a -> Integer -- E.g., Int to Integer
Prelude> :t fromIntegral
fromIntegral :: (Integral a, Num b) => a -> b -- Make more general
Prelude> 1 + 3.2
4.2
Prelude> (1 : Int) + 3.2
   * Non type-variable argument in the constraint: Fractional [a]
     (Use FlexibleContexts to permit this)
   * When checking the inferred type
     it :: forall a. (Num a, Fractional [a]) => [a]
Prelude> fromIntegral (1 :: Integer) + 3.2
4.2
Prelude> :t (/)
(/) :: Fractional a => a -> a -> a -- Non-integer division
The Show Typeclass

Show is helpful for debugging

Prelude> :t show
show :: Show a => a -> String
Prelude> show 3
"3"
Prelude> show 3.14159
"3.14159"
Prelude> show pi
"3.141592653589793"
Prelude> show True
"True"
Prelude> show (True, 3.14)
"(True,3.14)"
Prelude> show ["he","llo"]
"["he","llo"]"
The Read Typeclass

Simple parsing. You may need to tell it what type to look for

Prelude> :t read
read :: Read a => String -> a
Prelude> read "17" + 25
42

**Deduced type from context**

Prelude> read "4"
*** Exception: Prelude.read: no parse

**Not enough information**

Prelude> read "4" :: Int
4
Prelude> read "4" :: Integer
4
Prelude> read "4" :: Float
4.0
Prelude> read "(True, 42)" :: (Bool, Int)
(True,42)

**Tuples can be read**

Prelude> read "["hello","world"]" :: [String]
["hello","world"]

**Lists can be read**