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 $-2^{63}, ..., -1, 0, 1, ..., 2^{63} - 1$

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**.
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.
Type Annotations

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

For example,

- `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  
  \[
  \text{type String} = \text{[Char]}
  \]
- **Int** - fixed-precision integers
- **Integer** - arbitrary-precision integers
- **Float** - single-precision floating-point numbers
- **Double** - double-precision floating-point numbers
List Types

A list is sequence of values of the same type:

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

\[
[\text{’a’, ’b’, ’c’}] :: [\text{Char}]
\]

\[
\text{“abc”} :: [\text{Char}]
\]

\[
[[\text{True, True}], []] :: [[[\text{Bool}]]]
\]

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

A tuple is a sequence of values of different types:

\[
\begin{align*}
\text{(False, True)} & : : (\text{Bool}, \text{Bool}) \\
\text{(False, ‘a’, True)} & : : (\text{Bool}, \text{Char}, \text{Bool}) \\
\text{("Howdy", (True, 2))} & : : ([\text{Char}], (\text{Bool}, \text{Int}))
\end{align*}
\]

Note:
- \((t_1, t_2, \ldots, t_n)\) is the type of \(n\)-tuples whose \(i\)-th component has type \(t_i\) for any \(i\) in \(1 \ldots n\)
- 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 ($T_1$) to values of another type ($T_2$), with the type $T_1 \rightarrow T_2$

<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>not</td>
<td>$\text{Bool} \rightarrow \text{Bool}$</td>
</tr>
<tr>
<td>isDigit</td>
<td>$\text{Char} \rightarrow \text{Bool}$</td>
</tr>
<tr>
<td>toUpper</td>
<td>$\text{Char} \rightarrow \text{Char}$</td>
</tr>
<tr>
<td>(&amp;&amp;)</td>
<td>$\text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>$\text{(Int,Int)} \rightarrow \text{Int}$</td>
</tr>
<tr>
<td>add</td>
<td>$(x,y) = x+y$</td>
</tr>
<tr>
<td>zeroTo</td>
<td>$\text{Int} \rightarrow \text{[Int]}$</td>
</tr>
<tr>
<td>zeroTo</td>
<td>$n \rightarrow [0..n]$</td>
</tr>
</tbody>
</table>

One parameter functions!
Curried Functions

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
```

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 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 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: our first 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]`

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

> map (add' 1) [1,2,3]
[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.

\[
\text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}
\]

Means \( \text{Int} \rightarrow (\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})) \)

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

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

Means \( ((\text{mult} \ x) \ y) \ z \)
Polymorphic Functions

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.

- \texttt{length :: [a] \to \text{Int}}
  - for any type \(a\), \texttt{length} takes a list of values of type \(a\) and returns an integer

- \texttt{id :: a \to a}
  - for any type \(a\), \texttt{id} maps a value of type \(a\) to itself

- \texttt{head :: [a] \to a}
- \texttt{take :: \text{Int} \to [a] \to [a]}

\(a\) is a type variable
Polymorphic Types

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

<table>
<thead>
<tr>
<th>expression</th>
<th>polymorphic type</th>
<th>type variable bindings</th>
<th>resulting type</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>a -&gt; a</td>
<td>a=Int</td>
<td>Int -&gt; Int</td>
</tr>
<tr>
<td>id</td>
<td>a -&gt; a</td>
<td>a=Bool</td>
<td>Bool -&gt; Bool</td>
</tr>
<tr>
<td>length</td>
<td>[a] -&gt; Int</td>
<td>a=Char</td>
<td>[Char] -&gt; Int</td>
</tr>
<tr>
<td>fst</td>
<td>(a, b) -&gt; a</td>
<td>a=Char, b=Bool</td>
<td>Char</td>
</tr>
<tr>
<td>snd</td>
<td>(a, b) -&gt; b</td>
<td>a=Char, b=Bool</td>
<td>Bool</td>
</tr>
<tr>
<td>([], [])</td>
<td>([a], [b])</td>
<td>a=Char, b=Bool</td>
<td>([Char], [Bool])</td>
</tr>
</tbody>
</table>

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

What does the following function do, and what is its type?

\[
twice :: (t \rightarrow t) \rightarrow t \rightarrow t
\]

\[
twice \ f \ x = f \ (f \ x)
\]

> twice tail “abcd”
“cd”

What is the type of \( \text{twice twice} \)?

- The parameter and return type of \( \text{twice} \) are the same \((t \rightarrow t)\)
- Thus, \( \text{twice} \) and \( \text{twice twice} \) have the same type
- So, \( \text{twice twice} :: (t \rightarrow t) \rightarrow t \rightarrow t \)
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`
- `> sum [1.1,2.2,3.3]` → `6.6`
- `> sum ['a','b','c']` → **ERROR**

For:
- `a = Int`
- `a = Float`
- Char is not a numeric type
Class Constraints

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

\[ \text{id :: } t \rightarrow t \quad \text{length :: } [t] \rightarrow \text{Int} \]

This is when no operation is subjected to values of type \( t \)

What are the types of these functions?

\[ \text{min :: } \text{Ord a} \Rightarrow a \rightarrow a \rightarrow a \]
\[ \text{min } x \ y = \text{if } x < y \text{ then } x \text{ else } y \]

\[ \text{elem :: } \text{Eq a} \Rightarrow a \rightarrow [a] \rightarrow \text{Bool} \]
\[ \text{elem } x \ (y:ys) \quad | \quad x == y \quad = \quad \text{True} \]
\[ \text{elem } x \ (y:ys) \quad = \quad \text{elem } x \ ys \]
\[ \text{elem } x \ [] \quad = \quad \text{False} \]

Ord \( a \) and Eq \( a \) are class constraints

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

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

\[
\begin{align*}
\text{min} & : \text{Ord } a \Rightarrow a \rightarrow a \rightarrow a \\
\text{min } x \ y & = \text{if } x \ < \ y \ \text{then } x \ \text{else } y \\
\text{elem} & : \text{Eq } a \Rightarrow a \rightarrow [a] \rightarrow \text{Bool} \\
\text{elem } x \ (y:ys) \ | \ x \ == \ y & = \text{True} \\
\text{elem } x \ (y:ys) & = \text{elem } x \ ys \\
\text{elem } x \ [] & = \text{False}
\end{align*}
\]

Ord and Eq are type classes:

- **Num** (Numeric types)
  - (+) :: Num a \Rightarrow a \rightarrow a \rightarrow a
- **Eq** (Equality types)
  - (==) :: Eq a \Rightarrow a \rightarrow a \rightarrow \text{Bool}
- **Ord** (Ordered types)
  - (<) :: Ord a \Rightarrow a \rightarrow a \rightarrow \text{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]"

> 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?

- \( \text{second } xs \) = head (tail \( xs \))
- \( \text{swap } (x,y) \) = (\( y,x \))
- \( \text{pair } x \ y \) = (\( x,y \))
- \( \text{double } x \) = \( x*2 \)
- \( \text{palindrome } xs \) = reverse \( xs \) == \( xs \)
- \( \text{lessThanHalf } x \ y \) = \( x*2 < y \)

(3) Check your answers using GHCi.
second :: [a] -> a
second xs = head (tail xs)

swap :: (a, b) -> (b, a)
swap (x,y) = (y,x)

pair :: a -> b -> (a, b)
pair x y = (x,y)

double :: Num a => a -> a
double x = x*2

palindrome :: Eq a => [a] -> Bool
palindrome xs = reverse xs == xs

lessThanHalf :: (Num a, Ord a) => a -> a -> Bool
lessThanHalf x y = x * 2 < y