Polymorphism

Polymorphic Functions

doTwice :: (a -> a) -> a -> a

doTwice \( f \ x = f \ (f \ x) \)

*Operate* on different kinds values

```haskell
>>> double x = 2 * x
>>> yum x = x ++ " yum! yum!"

>>> doTwice double 10
40

>>> doTwice yum "cookie"
"cookie yum! yum!"```
QUIZ

What is the value of quiz?

greaterThan :: Int -> Int -> Bool
greaterThan x y = x > y

quiz = doTwice (greaterThan 10) 0

A. True
B. False
C. Type Error
D. Run-time Exception
E. 101

With great power, comes great responsibility!
```python
>>> doTwice (greaterThan 10) 0

36:9: Couldn't match type 'Bool' with 'Int'
    Expected type: Int -> Int
    Actual type: Int -> Bool
    In the first argument of 'doTwice', namely 'greaterThan 10'
    In the expression: doTwice (greaterThan 10) 0

The input and output types are different!

Cannot feed the output of (greaterThan 10 0) into greaterThan 10!
```

**Polymorphic Types**

But the type of doTwice would have spared us this grief.
>>> :t doTwice
doTwice :: (a -> a) -> a -> a

The signature has a *type parameter* t

- **re-use** doTwice to increment Int or concat String or ...
- The first argument f must take input t and return output t (i.e. t -> t)
- The second argument x must be of type t
- Then f x will also have type t ... and we can call f (f x).

But function is *incompatible* with doTwice

- if its input and output types *differ*

**QUIZ**
Let's make sure you're following!

What is the type of quiz?

\[ \text{quiz} \times f = f \times \]

A. \( a \rightarrow a \)
B. \( (a \rightarrow a) \rightarrow a \)
C. \( a \rightarrow b \rightarrow a \rightarrow b \)
D. \( a \rightarrow (a \rightarrow b) \rightarrow b \)
E. \( a \rightarrow b \rightarrow a \)

**QUIZ**

Let's make sure you're following!

What is the value of quiz?
apply :: a -> (a -> b) -> b
apply x f = f x

greaterThan :: Int -> Int -> Bool
greaterThan x y = x > y

quiz = apply 100 (greaterThan 10) :: Bool

A. Type Error
B. Run-time Exception
C. True
D. False
E. 110

Polymorphic Data Structures
Today, let's see **polymorphic data types**

which contain many kinds of values.

---

### Recap: Data Types

Recall that Haskell allows you to create brand new data types (03-haskell-types.html)

```haskell
data Shape
    = MkRect Double Double
    | MkPoly [(Double, Double)]
```
QUIZ

What is the type of MkRect?

data Shape
   = MkRect Double Double
   | MkPoly [(Double, Double)]

a. Shape

b. Double

c. Double -> Double -> Shape

d. (Double, Double) -> Shape

e. [(Double, Double)] -> Shape

Tagged Boxes
Values of this type are either two doubles tagged with Rectangle

````
>>> :type (Rectangle 4.5 1.2)
(Rectangle 4.5 1.2) :: Shape
```

or a list of pairs of Double values tagged with Polygon

````
ghci> :type (Polygon [(1, 1), (2, 2), (3, 3)])
(Polygon [(1, 1), (2, 2), (3, 3)]) :: Shape
```

Data values inside special Tagged Boxes

Datatypes are Boxed–and–Tagged Values

Recursive Data Types
We can define datatypes recursively too

```haskell
data IntList
    = INil                 -- ^ empty list
    | ICons Int IntList    -- ^ list with "hd" Int and "tl" IntList

deriving (Show)
```

(Ignore the bit about `deriving` for now.)

**QUIZ**

```haskell
data IntList
    = INil                 -- ^ empty list
    | ICons Int IntList    -- ^ list with "hd" Int and "tl" IntList

deriving (Show)
```

What is the type of `ICons`?

A. Int -> IntList -> List
B. IntList

C. Int -> IntList -> IntList

D. Int -> List -> IntList

E. IntList -> IntList

Constructing \textit{IntList}

Can only build \textit{IntList} via constructors.

\begin{verbatim}
>>> \textbf{type} INil
INil :: IntList

>>> \textbf{type} ICons
ICons :: Int -> IntList -> IntList
\end{verbatim}
**EXERCISE**

Write down a representation of type `IntList` of the list of three numbers 1, 2 and 3.

```plaintext
list_1_2_3 :: IntList
list_1_2_3 = ???

ICons 1 (ICons 2 (ICons 3 INil))
```

**Hint** Recursion means boxes within boxes

```
<table>
<thead>
<tr>
<th>ICons</th>
<th>ICons</th>
<th>ICons</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Recursively Nested Boxes
```

```
data IntTree =
| Leaf |
| Node Int IntTree IntTree
```
Trees: Multiple Recursive Occurrences

We can represent Int trees like

```haskell
data IntTree
    = ILeaf Int -- ^ single "leaf" w/ an Int
    | INode IntTree IntTree -- ^ internal "node" w/ 2 sub-trees
es

    deriving (Show)
```

A leaf is a box containing an Int tagged ILeaf e.g.

```haskell
>>> it1 = ILeaf 1
>>> it2 = ILeaf 2
```

A node is a box containing two sub-trees tagged INode e.g.

```haskell
>>> itt = INode (ILeaf 1) (ILeaf 2)
>>> itt' = INode itt itt
>>> INode itt' itt'
INode (INode (ILeaf 1) (ILeaf 2)) (INode (ILeaf 1) (ILeaf 2))
```
**Multiple Branching Factors**

e.g. 2-3 trees (http://en.wikipedia.org/wiki/2-3_tree)

data Int23T

\[
\begin{align*}
\text{Int23T} &= \text{ILeaf0} \\
\text{INode2} \text{Int } \text{Int23T } \text{Int23T} \\
\text{INode3} \text{Int } \text{Int23T } \text{Int23T } \text{Int23T}
\end{align*}
\]

deriving (Show)

An example value of type Int23T would be

\[
i23t :: \text{Int23T} \\
i23t = \text{INode3 } 0 \text{ t t t} \\
\text{where } t = \text{INode2 } 1 \text{ ILeaf0 ILeaf0}
\]

which looks like

Integer 2-3 Tree
Parameterized Types

We can define CharList or DoubleList - versions of IntList for Char and Double as

data CharList =
   CNil
| CCons Char CharList
  deriving (Show)

data DoubleList =
   DNil
| DCons Char DoubleList
  deriving (Show)
Don’t Repeat Yourself!

Don’t repeat definitions - Instead *reuse* the list *structure* across *all* types!

Find abstract *data* patterns by

- identifying the *different* parts and
- refactor those into *parameters*

A Refactored List

Here are the three types: What is common? What is different?

```haskell
data IList = INil | ICons Int     IList

data CList = CNil | CCons Char    CList

data DList = DNil | DCons Double DList
```

**Common:** Nil/Cons structure

**Different:** type of each “head” element

Refactored using Type Parameter
data List a = Nil | Cons a (List a)

Recover original types as instances of List

type IntList = List Int
type CharList = List Char
type DoubleList = List Double

Polymorphic Data has Polymorphic Constructors

Look at the types of the constructors

>>> :type Nil
Nil :: List a
That is, the Empty tag is a value of any kind of list, and

```haskell
>>> :type Cons
Cons :: a -> List a -> List a
```

Cons takes an a and a List a and returns a List a.

cList :: List Char  -- list where 'a' = 'Char'
cList = Cons 'a' (Cons 'b' (Cons 'c' Nil))

iList :: List Int   -- list where 'a' = 'Int'
iList = Cons 1 (Cons 2 (Cons 3 Nil))

dList :: List Double -- list where 'a' = 'Double'
dList = Cons 1.1 (Cons 2.2 (Cons 3.3 Nil))

---

**Polymorphic Function over Polymorphic**
Data

Let's write the list length function

\[
\text{len} :: \text{List } a \rightarrow \text{Int} \\
\text{len } \text{Nil} = 0 \\
\text{len } (\text{Cons } x \text{ xs}) = 1 + \text{len } \text{xs}
\]

\text{len} doesn't care about the actual values in the list - only “counts” the number of \text{Cons} constructors

Hence \text{len} :: \text{List } a \rightarrow \text{Int}

- we can call \text{len} on any kind of list.

```\text{Haskell}
>>> \text{len } [1.1, 2.2, 3.3, 4.4] \quad -- \ a := \text{Double} \\
4

>>> \text{len } "\text{mmm donuts!}" \quad -- \ a := \text{Char} \\
11

>>> \text{len } [[1], [1,2], [1,2,3]] \quad -- \ a := \ ??? \\
3
```
**Built-in Lists?**

This is exactly how Haskell’s “built-in” lists are defined:

```haskell
data [a] = [] | (:) a [a]
```

```haskell
data List a = Nil | Cons a (List a)
```

- Nil is called `[`]
- Cons is called `:`

Many list manipulating functions e.g. in [Data.List][1] are *polymorphic* - Can be reused across all kinds of lists.

```haskell
(++) :: [a] -> [a] -> [a]
head :: [a] -> a
tail :: [a] -> [a]
```
Generalizing Other Data Types

Polymorphic trees

data Tree a
  = Leaf a
  | Node (Tree a) (Tree a)
  deriving (Show)

Polymorphic 2–3 trees

data Tree23 a
  = Leaf0
  | Node2 (Tree23 a) (Tree23 a)
  | Node3 (Tree23 a) (Tree23 a) (Tree23 a)
  deriving (Show)
Kinds

List :\ a\ corresponds\ to\ lists\ of\ values\ of\ type\ :\ a:.\n
If :\ a:\ is\ the\ type\ parameter,\ then\ what\ is\ List?\

A type-constructor\ that\ –\ takes\ as\ input\ a\ type\ :\ a:\ –\ returns\ as\ output\ the\ type\ List :\ a:

But\ wait,\ if\ List\ is\ a\ type-constructor\ then\ what\ is\ its\ “type”?\

- A kind is the “type” of a type.

>>> :kind Int
Int :: *

>>> :kind Char
Char :: *

>>> :kind Bool
Bool :: *

Thus, List is a function from any “type” to any other “type”, and so

>>> :kind List
List :: * -> *
QUIZ

What is the kind of ->? That is what does GHCi say if we type

>>> :kind (->)

A. *
B. * -> *
C. * -> * -> *

We will not dwell too much on this now.

As you might imagine, they allow for all sorts of abstractions over data.

If interested, see this for more information about kinds (http://en.wikipedia.org/wiki/Kind_(type_theory)).
Bottling Computation Patterns

Polymorphism and HOFs are the Secret Sauce

Refactor arbitrary repeated code patterns ...

... into precisely specified and reusable functions

EXERCISE: Iteration

Write a function that squares a list of Int

squares :: [Int] -> [Int]
squares ns = ???

When you are done you should see
Pattern: Iteration

Next, let's write a function that converts a String to uppercase.

```haskell
>>> shout "hello"
"HELLO"
```

Recall that in Haskell, a String is just a [Char].

```haskell
shout :: [Char] -> [Char]
shout = ???
```

Hoogle (http://haskell.org/hoogle) to see how to transform an individual Char.
-- rename 'squares' to 'foo'
foo []  = []
foo (x:xs) = (x * x)  : foo xs

-- rename 'shout' to 'foo'
foo []  = []
foo (x:xs) = (toUpper x) : foo xs

Step 2 Identify what is different

- In squares we transform x to x * x
- In shout we transform x to Data.Char.toUpper x

Step 3 Make differences a parameter

- Make transform a parameter f

foo f []  = []
foo f (x:xs) = (f x) : foo f xs

Done We have bottled the computation pattern as foo (aka map)

map f []  = []
map f (x:xs) = (f x) : map f xs

map bottles the common pattern of iteratively transforming a list:

Fairy In a Bottle
QUIZ

What is the type of \texttt{map}?

\texttt{map} :: \ ???
\texttt{map f \[]} = \[]
\texttt{map f \(x:xs\)} = (f x) : \texttt{map f xs}

A. \((\text{Int} \to \text{Int}) \to [\text{Int}] \to [\text{Int}]\)
B. \((\text{a} \to \text{a}) \to [\text{a}] \to [\text{a}]\)
C. \([\text{a}] \to [\text{b}]\)
D. \((\text{a} \to \text{b}) \to [\text{a}] \to [\text{b}]\)
E. \((\text{a} \to \text{b}) \to [\text{a}] \to [\text{a}]\)
The type precisely describes \textit{map}

\begin{verbatim}
>>> :type map
map :: (a -> b) -> [a] -> [b]
\end{verbatim}

That is, \textit{map} takes two inputs

- a \textit{transformer} of type \textit{a} -> \textit{b}
- a \textit{list} of values \textit{[a]}

and it returns as output

- a \textit{list} of values \textit{[b]}

that can only come by applying \textit{f} to each element of the input list.

\section*{Reusing the Pattern}

Lets reuse the pattern by \textit{instantiating} the transformer
EXERCISE

Suppose I have the following type

```haskell
type Score = (Int, Int) -- pair of scores for Hw0, Hw1
```

Use `map` to write a function

```haskell
total :: [Score] -> [Int]
total xs = map (???) xs
```

such that

```haskell
>>> total [(10, 20), (15, 5), (21, 22), (14, 16)]
[30, 20, 43, 30]
```
The Case of the Missing Parameter

Note that we can write `shout` like this

```
shout :: [Char] -> [Char]
shout = map Char.toUpper
```

Huh. No parameters? Can someone explain?

The Case of the Missing Parameter

In Haskell, the following all mean the same thing

Suppose we define a function

```
add :: Int -> Int -> Int
add x y = x + y
```

Now the following all mean the same thing
plus x y = add x y
plus x  = add x
plus     = add

Why? *equational reasoning*! In general

foo x = e x

-- is equivalent to

foo = e

as long as x doesn’t appear in e.

Thus, to save some typing, we *omit* the extra parameter.

---

**Pattern: Reduction**

Computation patterns are *everywhere* lets revisit our old `sumList`