BAHUG 101 - Lecture 2

23th September 2015

Outline of Today's Lecture

- Parametric Polymorphism
- Total and Partial functions
- Recursion Patterns
- Functional Programming style
- Currying and Partial Application

Parametric Polymorphism

How the type **a** is determined

Polymorphic functions have “type variables” in their type definition.

```
lengthList :: [a] → Integer
lengthList [] = 0
lengthList (x:xs) = 1 + lengthList xs
```

How the type **a** is determined

Polymorphic functions have “type variables” in their type definition.

```
lengthList :: [a] → Integer
lengthList [] = 0
lengthList (x:xs) = 1 + lengthList xs
```

In Haskell, the *caller of a function gets to determine the type* when creaing a polymorphic function.

Functions that assume inputs are impossible

Take the following function.

```
bogus :: [a] -> Bool
bogus ('X' : _) = True
bogus _         = False
```

[1] We can do something like this with ad hoc polymorphism, type families, GADTs

Functions that assume inputs are impossible

Take the following function.

```
bogus :: [a] -> Bool
bogus ('X' : _) = True
bogus _         = False
```

It assumes `[a]` is `[Char]` in the definition of the function, and it thus illegal. We can do something if `a` is an `Int` and something different if `a` is a `Char`.[1]

[1] We can do something like this with ad hoc polymorphism, type families, GADTs

Functions that work for any input are ok

In the following function, we do not need to know what the list contains to determine if it is empty.

```
notEmpty :: [a] -> Bool
notEmpty (_:_) = True
notEmpty []    = False
```

Parameticity allows for type erasure

During compilation the types are removed from the code. They are not needed during execution because the types are known at compile time!

Can we write this function?

```
strange :: a -> b
```

Can we write this function?

```haskell
strange :: a -> b

strange = error "impossible!" -- error :: String -> a
```

There is no way to write this function! It would need to work for any `a` and any `b`.

Can we write this function?

Given the type signature, do we know how to write this function?

```haskell
limited :: a -> a
```

[2]You can programmatically do this with a Haskell package called (Djinn)[http://lambda-the-ultimate.org/node/1178]

Can we write this function?

Given the type signature, do we know how to write this function?

```haskell
limited :: a -> a

limited x = x
```

We know that **`limited`** must be the identity function because it is the only function, for any **`a`**, that takes an **`a`** and returns an **`a`**.[2]

[2] You can programmatically do this with a Haskell package called (Djinn)[http://lambda-the-ultimate.org/node/1178]

Partial and Total Functions

What happens in this example?

To take the first element of a list, you could use the **head** function.

```
head :: [a] -> a
head (x:_) = x
```

What does `head []` produce?

What happens in this example?

To take the first element of a list, you could use the **head** function.

```haskell
head :: [a] -> a
head (x:_) = x
```

What does `head []` produce?

An error! It cannot produce a value of type `a`.

What happens in this example?

To take the first element of a list, you could use the **head** function.

```
head :: [a] → a
head (x:_) = x
```

What does `head []` produce?

An error! It cannot produce a value of type `a`.

The **head** in Haskell looks like this.

```
head :: [a] → a
head (x:_) = x
head [] = errorEmptyList "head"
```

head is a partial function

head is a *partial function*; it is not defined for all inputs. Certain inputs will cause **head** to crash.

`head` is a partial function

`head` is a *partial function*; it is not defined for all inputs. Certain inputs will cause `head` to crash.

In contrast, a *total function* is a function defined for all inputs.

Partial Functions should be avoided

It is a common Haskell practice to avoid partial functions, such as

- `head`
- `tail`
- `init`
- `last`
- `(!!)`

How to avoid partial functions

Here is a total function using partial functions. It is a bit cludgey.

```
doStuff1 :: [Int] -> Int
doStuff1 []  = 0
doStuff1 [_] = 0
doStuff1 xs  = head xs + (head (tail xs))
```

How to avoid partial functions

Here is a total function using partial functions. It is a bit cludgey.

```
doStuff1 :: [Int] -> Int
doStuff1 []  = 0
doStuff1 [_] = 0
doStuff1 xs  = head xs + (head (tail xs))
```

We can make it simpler by pattern matching.

```
doStuff2 :: [Int] -> Int
doStuff2 []          = 0
doStuff2 [_]         = 0
doStuff2 (x1:x2:_) = x1 + x2
```

Recursion Patterns

What can we do with `[a]`?

If we have a function taking in a value of type `[a]`, what can we do to it?

- Do something to every element of the list.

What can we do with `[a]`?

If we have a function taking in a value of type `[a]`, what can we do to it?

- Do something to every element of the list.
- Keep only some of the elements of the list (based on some test).

What can we do with `[a]`?

If we have a function taking in a value of type `[a]`, what can we do to it?

- Do something to every element of the list.
- Keep only some of the elements of the list (based on some test).
- Combine all the elements of the list in some form.

What can we do with `[a]`?

If we have a function taking in a value of type `[a]`, what can we do to it?

- Do something to every element of the list.
- Keep only some of the elements of the list (based on some test).
- Combine all the elements of the list in some form.
- There are other things. What can you think of?

Do something to every element of a list : add

Here is a simple function that adds one to every element in a list of integers.

```
addOneToAll :: [Int] -> [Int]
addOneToAll []     = []
addOneToAll (x:xs) = x + 1 : addOneToAll xs
```

Do something to every element of a list : absolute value

Here is a simple function that takes the absolute value of every element in a list.

```
absAll :: [Int] → [Int]
absAll []     = []
absAll (x:xs) = abs x : absAll xs
```

Do something to every element of a list : square

Here is a simple function that squares all of the elements in a list.

```
squareAll :: [Int] -> [Int]
squareAll []     = []
squareAll (x:xs) = x^2 : squareAll xs
```

Notice a pattern?

It seems we keep writing the following.

```
doSomethingToEachInt :: [Int] -> [Int]
doSomethingToEachInt []     = []
doSomethingToEachInt (x:xs) = ? x : doSomethingToEachInt xs
```

where

```
f :: Int -> Int
```

Pass in the function on `Int`s

If we pass in the function that works on `Int`s we can simplify `doSomethingToEachInt`.

```
doSomethingToEachInt' :: (Int → Int) → [Int] → [Int]
doSomethingToEachInt' _ []     = []
doSomethingToEachInt' f (x:xs) = f x : doSomethingToEachInt' f xs
```

Map

We can make `doSomethingToEachInt` even more generic to work on lists of any type if the `f` we pass in works for any input `a` to any output `b`.

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

Redoing our original functions

We can rewrite our original functions as so.

```
addOneToAll' xs = map (+1) xs
```

Redoing our original functions

We can rewrite our original functions as so.

```
addOneToAll' xs = map (+1) xs

absAll' xs = map abs xs
```

Redoing our original functions

We can rewrite our original functions as so.

```
addOneToAll' xs = map (+1) xs

absAll' xs = map abs xs

squareAll' xs = map (^2) xs
```

We can also ignore elements in a list

What if we only want to keep the positive integers?

```
keepOnlyPositive :: [Int] -> [Int]
keepOnlyPositive [] = []
keepOnlyPositive (x:xs)
  | x > 0     = x : keepOnlyPositive xs
  | otherwise = keepOnlyPositive xs
```

We can also ignore elements in a list

What if we only want to keep the positive integers?

```haskell
keepOnlyPositive :: [Int] -> [Int]
keepOnlyPositive [] = []
keepOnlyPositive (x:xs)
  | x > 0     = x : keepOnlyPositive xs
  | otherwise = keepOnlyPositive xs
```

Or only the even values?

```haskell
keepOnlyEven :: [Int] -> [Int]
keepOnlyEven [] = []
keepOnlyEven (x:xs)
  | even x    = x : keepOnlyEven xs
  | otherwise = keepOnlyEven xs
```

Filter

We see a similar abstraction.

```
keepSomething :: (Int → Bool) → [Int] → [Int]
keepSomething _ [] = []
keepSomething p (x:xs)
  | p x       = x : keepSomething p xs
  | otherwise = keepSomething p xs
```

Filter

We see a similar abstraction.

```haskell
keepSomething :: (Int -> Bool) -> [Int] -> [Int]
keepSomething _ [] = []
keepSomething p (x:xs)
  | p x       = x : keepSomething p xs
  | otherwise = keepSomething p xs
```

Similar to before, we can abstract this to lists of any type.

```haskell
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
  | p x = x : filter p xs     -- Keep the element
  | otherwise = filter p xs -- Ignore an element
```

`p` is known as a *predicate* function.

Can we abstract combining elements?

We can combine elements of a list. Take for example the following functions.

```
sum' :: [Int] -> Int
sum' []     = 0
sum' (x:xs) = x + sum' xs
```

Can we abstract combining elements?

We can combine elements of a list. Take for example the following functions.

```
sum' :: [Int] -> Int
sum' []      = 0
sum' (x:xs) = x + sum' xs

product' :: [Int] -> Int
product' [] = 1
product' (x:xs) = x * product' xs
```

Can we abstract combining elements?

We can combine elements of a list. Take for example the following functions.

```
sum' :: [Int] -> Int
sum' []     = 0
sum' (x:xs) = x + sum' xs

product' :: [Int] -> Int
product' [] = 1
product' (x:xs) = x * product' xs

length' :: [a] -> Int
length' []     = 0
length' (_:xs) = 1 + length' xs
```

The combining case basic formula

The basic formula is this.

```
combine :: [a] -> b
combine [] = someBaseValue
combine (x:xs) = x `binaryFunction` combine xs
```

Fold

The basic formula can be written in Haskell as such.

```haskell
fold :: (a -> b -> b) -> b  -> [a] -> b
fold f z []      = z -- base value
fold f z (x:xs) = f  x  (fold f z xs)
-- types             a       b
```

This function has another name, **`foldr`**, in the standard Prelude.

Rewriting our examples

Our functions from before can be written in a simpler manner using `fold`

```
sum''     = fold (+) 0
product'' = fold (*) 1
length''  = fold addOne 0
  where addOne _ s = 1 + s
```

Different types of folds

In Haskell, there are several different common kinds of folds.

- `foldr`, which folds from the right.

```
foldr f z [a,b,c] == a `f` (b `f` (c `f` z))
```

Different types of folds

In Haskell, there are several different common kinds of folds.

- `foldr`, which folds from the right.

```
foldr f z [a,b,c] == a `f` (b `f` (c `f` z))
```

- `foldl`, which folds from the left.

```
foldl f z [a,b,c] == ((z `f` a) `f` b) `f` c
```

Different types of folds

In Haskell, there are several different common kinds of folds.

- `foldr`, which folds from the right.

```
foldr f z [a,b,c] == a `f` (b `f` (c `f` z))
```

- `foldl`, which folds from the left.

```
foldl f z [a,b,c] == ((z `f` a) `f` b) `f` c
```

- `foldr1`, which folds from the right *eagerly*.

Different types of folds

In Haskell, there are several different common kinds of folds.

- `foldr`, which folds from the right.

```
foldr f z [a,b,c] == a `f` (b `f` (c `f` z))
```

- `foldl`, which folds from the left.

```
foldl f z [a,b,c] == ((z `f` a) `f` b) `f` c
```

- `foldr1`, which folds from the right *eagerly*.
- `foldl1`, which folds from the left *eagerly*.

Different types of folds

In Haskell, there are several different common kinds of folds.

- `foldr`, which folds from the right.

```
foldr f z [a,b,c] == a `f` (b `f` (c `f` z))
```

- `foldl`, which folds from the left.

```
foldl f z [a,b,c] == ((z `f` a) `f` b) `f` c
```

- `foldr1`, which folds from the right *eagerly*.
- `foldl1`, which folds from the left *eagerly*.

In general, `foldl` has poor performance. Use `foldr` or `foldl1` instead.

Functional Programming

Functional Combinators

It is common in Haskell to "glue" functions together.

An example of this is the `(.)` *compose* combinator.

```
(.) :: (b → c) → (a → b) → a → c
f g x = f (g x)
```

Functional Combinators

It is common in Haskell to “glue” functions together.

An example of this is the `(.)` *compose* combinator.

```
(.) :: (b → c) → (a → b) → a → c
f g x = f (g x)
```

If we ant to both add one and multiply by 4 for each element in a list, we can do it this way.

```
add1Mul4 :: [Int] → [Int]
add1Mul4 x = map ((*4) . (+1)) x
```

Function Application Combinator

Another interesting combinator is `($)`

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

Function Application Combinator

Another interesting combinator is `($)`

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

This function is often used to remove parenthesis (due to operator/function precidence). For example

```
negateNumEven1 :: [Int] -> Int
negateNumEven1 x = negate (length (filter even x))
```

Function Application Combinator

Another interesting combinator is `($)`

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

This function is often used to remove parenthesis (due to operator/function precidence). For example

```
negateNumEven1 :: [Int] -> Int
negateNumEven1 x = negate (length (filter even x))
```

can be rewritten as

```
negateNumEven2 :: [Int] -> Int
negateNumEven2 x = negate $ length $ filter even x
```

Function Application Combinator

Another interesting combinator is `($)`

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

This function is often used to remove parenthesis (due to operator/function precidence). For example

```
negateNumEven1 :: [Int] -> Int
negateNumEven1 x = negate (length (filter even x))
```

can be rewritten as

```
negateNumEven2 :: [Int] -> Int
negateNumEven2 x = negate $ length $ filter even x
```

or as

```
negateNumEven3 :: [Int] -> Int
negateNumEven3 x = negate . length . filter even $ x
```

Lambda expressions

Lambda expressions allow us to define small functions inline. For example

```
duplicate1 :: [String] -> [String]
duplicate1 = map dup
  where dup x = x ++ x
```

Lambda expressions

Lambda expressions allow us to define small functions inline. For example

```
duplicate1 :: [String] -> [String]
duplicate1 = map dup
  where dup x = x ++ x
```

can be simplified as

```
duplicate2 :: [String] -> [String]
duplicate2 = map (\x -> x ++ x)
```

Lambda expressions are best used for only the smallest functions. Otherwise use a helper function.

Currying and Partial Application

Does the multiple input functions look strange?

When we have a function that takes multiple inputs, we didn't discuss the syntax too much. Why are all but the last types inputs, and the last one the output?

```
f :: Int -> Int -> Int
f x y = 2*x + y
```

Does the multiple input functions look strange?

When we have a function that takes multiple inputs, we didn't discuss the syntax too much. Why are all but the last types inputs, and the last one the output?

```haskell
f :: Int -> Int -> Int
f x y = 2*x + y
```

In truth, all Haskell functions *take only one input*. When written out, `f` looks like so.

```haskell
f' :: Int -> (Int -> Int) -- Takes in an Int, returns a function
f' x = \y -> 2*x + y
```

Does the multiple input functions look strange?

When we have a function that takes multiple inputs, we didn't discuss the syntax too much. Why are all but the last types inputs, and the last one the output?

```
f :: Int -> Int -> Int
f x y = 2*x + y
```

In truth, all Haskell functions *take only one input*. When written out, `f` looks like so.

```
f' :: Int -> (Int -> Int) -- Takes in an Int, returns a function
f' x = \y -> 2*x + y
```

Function application is left associative, so the following are equivalent.

```
f 3 2 = (f 3) 2
```

Currying

The concept of representing multi-argument functions as single argument ones is known as *currying*.

Currying

The concept of representing multi-argument functions as single argument ones is known as *currying*.

We can make a function take two arguments by instead taking one pair.

```
f'' :: (Int, Int) -> Int
f'' (x, y) = 2*x + y
```

Currying

The concept of representing multi-argument functions as single argument ones is known as *currying*.

We can make a function take two arguments by instead taking one pair.

```
f'' :: (Int, Int) -> Int
f'' (x, y) = 2*x + y
```

And convert between these forms using the `curry` and `uncurry` functions.

```
curry :: ((a,b) -> c) -> a -> b -> c
curry f x y = f (x,y)

uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (x,y) = f x y
```

Partial Application

Since all functions in Haskell only really take in one input and potentially return a function, we can choose to only apply some of the arguments.

```
add x y = x + y
add4 y = add 4 y
```

This is called *partial application*. This only works for applying arguments from left to right order.

Wholemeal Programming

Consider the following function.

```
foobar :: [Integer] -> Integer
foobar []       = 0
foobar (x:xs)
  | x > 3       = (7*x + 2) + foobar xs
  | otherwise = foobar xs
```

It isn't very Haskell-y because it does a lot in one function and is works at a low-level.

Wholemeal Programming

Consider the following function.

```haskell
foobar :: [Integer] -> Integer
foobar []      = 0
foobar (x:xs)
  | x > 3      = (7*x + 2) + foobar xs
  | otherwise = foobar xs
```

It isn't very Haskell-y because it does a lot in one function and is works at a low-level.

Instead, a Haskell programmer would probably write this.

```haskell
foobar' :: [Integer] -> Integer
foobar' = sum . map ((+2) . (*7)) . filter (>3)
—                -----------------   ----------
—                partially applied   partially applied
```

Instead of thinking of direct manipulations, we can think of what kind of processing "pipeline" we want.