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 = ???
```

Hoole (http://haskell.org/hoole) to see how to transform an individual Char
Iteration

Common strategy: \textit{iteratively} transform each element of input list

Like humans and monkeys, shout and squares share 93\% of their DNA (http://www.livescience.com/health/070412_rhesus_monkeys.html)

Super common \textit{computation pattern}!

Abstract Iteration “Pattern” into Function

Remember D.R.Y. (Don’t repeat yourself)

\textbf{Step 1} Rename all variables to remove accidental \textit{differences}
-- rename 'squares' to 'foo'
foo [] = []
foo (x:xs) = (x * x) : foo xs

-- rename 'shout' to 'foo'
foo [] = []
foo (x:xs) = (Data.Char.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 `map`?

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

A. \((\text{Int} \to \text{Int}) \to \text{[Int]} \to \text{[Int]}\)

B. \((a \to a) \to \text{[a]} \to \text{[a]}\)

C. \([a] \to [b]\)

D. \((a \to b) \to \text{[a]} \to \text{[b]}\)

E. \((a \to b) \to \text{[a]} \to \text{[a]}\)
The type precisely describes \texttt{map}

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

That is, \texttt{map} takes two inputs

- a \textit{transformer} of type \( a \to b \)
- a \textit{list} of values \([a]\)

and it returns as output

- a list of values \([b]\)

that can only come by applying \( f \) to each element of the input list.

\section*{Reusing the Pattern}

Lets reuse the pattern by \textit{instantiating} the transformer
**shout**

-- OLD with recursion

shout :: [Char] -> [Char]
shout [] = []
shout (x:xs) = Char.toUpper x : shout xs

-- NEW with map

shout :: [Char] -> [Char]
shout xs = map (???) xs

**squares**

-- OLD with recursion

squares :: [Int] -> [Int]
squares [] = []
squares (x:xs) = (x * x) : squares xs

-- NEW with map

squares :: [Int] -> [Int]
squares xs = map (???) xs
**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

```haskell
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

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

Now the following all mean the same thing
\[
\begin{align*}
\text{plus } x \ y &= \text{add } x \ y \\
\text{plus } x &= \text{add } x \\
\text{plus} &= \text{add}
\end{align*}
\]

Why? *equational reasoning!* In general

\[
\text{foo } x = e \ x
\]

--- is equivalent to

\[
\text{foo } = e
\]

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

Next, a function that *concatenates* the String s in a list

catList :: [String] -> String
catList [] = ""
catList (x:xs) = x ++ (catList xs)

*Let's spot the pattern!*

**Step 1 Rename**

foo [] = 0
foo (x:xs) = x + foo xs

foo [] = ""
foo (x:xs) = x ++ foo xs
Step 2 Identify what is different

1. ???
2. ???

Step 3 Make differences a parameter

\[
\text{foo } p_1 \ p_2 \ [ ] = ??? \\
\text{foo } p_1 \ p_2 \ (x:xs) = ???
\]

**EXERCISE: Reduction/Folding**

This pattern is commonly called reducing or folding

\[
\text{foldr} :: (a -> b -> b) -> b -> [a] -> b \\
\text{foldr op base } [] = base \\
\text{foldr op base } (x:xs) = \text{op } x \ (\text{foldr op base } xs)
\]

Can you figure out how \texttt{sumList} and \texttt{catList} are just instances of \texttt{foldr}?

\[
\text{\texttt{sumList} = \texttt{foldr (+) 0}} \\
\text{\texttt{catList} = \texttt{foldr (++) ""}}
\]
sumList :: [Int] -> Int
sumList xs = foldr (?op) (?base) xs

catList :: [String] -> String
catList xs = foldr (?op) (?base) xs

**Executing `foldr`**

To develop some intuition about `foldr` lets “run” it a few times by hand.
foldr op b (a1:a2:a3:a4:[])
==> 
a1 `op` (foldr op b (a2:a3:a4:[]))
==>
a1 `op` (a2 `op` (foldr op b (a3:a4:[])))
==>
a1 `op` (a2 `op` (a3 `op` (foldr op b (a4:[]))))
==>
a1 `op` (a2 `op` (a3 `op` (a4 `op` foldr op b [])))
==> 
a1 `op` (a2 `op` (a3 `op` (a4 `op` b)))

Look how it mirrors the structure of lists!

- (:) is replaced by op
- [] is replaced by base

So

foldr (+) 0 (x1:x2:x3:x4:[])
==> x1 + (x2 + (x3 + (x4 + 0)))
Typing `foldr`

`foldr :: (a -> b -> b) -> b -> [a] -> b`

`foldr op base []     = base`

`foldr op base (x:xs) = op x (foldr op base xs)`

`foldr` takes as input

- a *reducer* function of type `a -> b -> b`
- a *base* value of type `b`
- a *list* of values to reduce `[a]`

and returns as output

- a *reduced* value `b`

\[ (a \to b \to b) \to b \to [a] \to b \]

\[ \text{foldr } op \ b \left( a_1 : a_2 : a_3 : a_4 : [] \right) \]

\[ \left( a \circ (a_2 \circ (a_3 \circ (a_4 \circ b))) \right) \]

\[ z :: a \to b \to b \]

**QUIZ**

Recall the function to compute the `len` of a list
len :: [a] -> Int
len [] = 0
len (x:xs) = 1 + len xs

Which of these is a valid implementation of \texttt{len}?

\begin{itemize}
  \item [A.] \texttt{len} = \texttt{foldr (\n -> n + 1) 0}
  \item [B.] \texttt{len} = \texttt{foldr (\n m -> n + m) 0}
  \item [C.] \texttt{len} = \texttt{foldr (\_ n -> n + 1) 0}
  \item [D.] \texttt{len} = \texttt{foldr (\x xs -> 1 + len xs) 0}
  \item [E.] All of the above
\end{itemize}

\textbf{The Missing Parameter Revisited}

We wrote \texttt{foldr} as

\[
\begin{array}{c}
  x_1 : x_2 : (x_3 : (x_4 : [])) \\
  \downarrow \\
  (1 + (1 + (1 + (1 + 0)))) \\
  \downarrow \\
  (\upsilon \rightarrow 1 + \upsilon)
\end{array}
\]
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr op base [] = base
foldr op base (x:xs) = op x (foldr op base xs)

but can also write this

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr op base = go
  where
    go [] = base
    go (x:xs) = op x (go xs)

Can someone explain where the xs went missing?

**Trees**

Recall the Tree a type from last time
**Some Functions on Trees**

Let's write a function to compute the height of a tree.
height :: Tree a -> Int
height Leaf = 0
height (Node x l r) = 1 + max (height l) (height r)

Here's another to sum the leaves of a tree:

sumTree :: Tree Int -> Int
sumTree Leaf = ???
sumTree (Node x l r) = ???

Gathers all the elements that occur as leaves of the tree:

toList :: Tree a -> [a]
toList Leaf = ???
toList (Node x l r) = ???

Let's give it a whirl

>>> height tree123
2

>>> sumTree tree123
6

>>> toList tree123
[1,2,3]
**Pattern: Tree Fold**

Can you spot the pattern? Those three functions are almost the same!

**Step 1: Rename to maximize similarity**

```haskell
-- height
foo Leaf = 0
foo (Node x l r) = 1 + max (foo l) (foo l)

-- sumTree
foo Leaf = 0
foo (Node x l r) = foo l + foo r

-- toList
foo Leaf = []
foo (Node x l r) = x : foo l ++ foo r
```

**Step 2: Identify the differences**

1. ???
2. ???

**Step 3 Make differences a parameter**

```haskell
foo p1 p2 Leaf = ???
foo p1 p2 (Node x l r) = ???
```
**Pattern: Folding on Trees**

\[
\text{tFold } \text{op } b \text{ Leaf} = b \\
\text{tFold } \text{op } b \text{ (Node } x \text{ l r)} = \text{op } \times \text{ (tFold } \text{op } b \text{ l)} (\text{tFold } \text{op } b \text{ r)}
\]

Let's try to work out the type of tFold!

\[
\text{tFold} :: \text{t}_\text{op} \rightarrow \text{t}_\text{b} \rightarrow \text{Tree } a \rightarrow \text{t}_\text{out}
\]
QUIZ

Suppose that t :: Tree Int.

What does \( \text{tFold } (\lambda x y z \rightarrow y + z) \) 1 t return?

a. 0  
b. the largest element in the tree t  
c. the height of the tree t  
d. the number-of-leaves of the tree t  
e. type error

EXERCISE

Write a function to compute the largest element in a tree or 0 if tree is empty or all negative.
Map over Trees

We can also write a tmap equivalent of map for Trees

```haskell
treeMap :: (a -> b) -> Tree a -> Tree b
treeMap f (Leaf x) = Leaf (f x)
treeMap f (Node l r) = Node (treeMap f l) (treeMap f r)
```

which gives

```haskell
>>> treeMap (\n -> n * n) tree123 -- square all elements of tree
Node 1 (Node 4 Leaf Leaf) (Node 9 Leaf Leaf)
```
EXERCISE

Recursion is HARD TO READ do we really have to use it?

Let's rewrite TreeMap using tFold!

```haskell
treeMap :: (a -> b) -> Tree a -> Tree b
treeMap f t = tFold op base t
    where
        op   = ???
        base = ???
```

When you are done, we should get

```haskell
>>> animals = Node "cow" (Node "piglet" Leaf Leaf) (Leaf "hippo" Leaf Leaf)
>>> treeMap reverse animals
Node "woc" (Node "telgip" Leaf Leaf) (Leaf "oppih" Leaf Leaf)
```
Examples: `foldDir`

```haskell
data Dir a
  = Fil a               -- ^ A single file named `a`
  | Sub a [Dir a]       -- ^ A sub-directory name `a` with contents `[Dir a]`

data DirElem a
  = SubDir a            -- ^ A single Sub-Directory named `a`
  | File a               -- ^ A single File named `a`

foldDir :: ([a] -> r -> DirElem a -> r) -> r -> Dir a -> r
foldDir f r0 dir = go [] r0 dir
  where
    go stk r (Fil a)      = f stk r (File a)
    go stk r (Sub a ds)   = L.foldl' (go stk') r' ds
      where
        r'     = f stk r (SubDir a)
        stk'   = a:stk
```

`foldDir` takes as input

- an accumulator `f` of type `[a] -> r -> DirElem a -> r`
  - takes as input the path `[a]`, the current result `r`, the next `DirElem [a]`
  - and returns as output the new result `r`
• an initial value of the result $r_0$ and
• directory to fold over dir

And returns the result of running the accumulator over the whole dir.

Examples: Spotting Patterns In The “Real” World

These patterns in “toy” functions appear regularly in “real” code

1. Start with beginner’s version riddled with explicit recursion (swizzle-v0.html).

2. Spot the patterns and eliminate recursion using HOFs (swizzle-v1.html).

3. Finally refactor the code to “swizzle” and “unswizzle” without duplication (swizzle-v2.html).

Try it yourself

• Rewrite the code that swizzes Char to use the Map k v type in Data.Map
Which is more readable? HOFs or Recursion

At first, recursive versions of `shout` and `squares` are easier to follow

- `fold` takes a bit of getting used to!

With practice, the higher-order versions become easier

- only have to understand specific operations
- recursion is lower-level & have to see “loop” structure
- worse, potential for making silly off-by-one errors

Indeed, HOFs were the basis of `map/reduce` and the big-data revolution (http://en.wikipedia.org/wiki/MapReduce)

- Can parallelize and distribute computation patterns just once
  (https://www.usenix.org/event/osdi04/tech/full_papers/dean/dean.pdf)

- Reuse (http://en.wikipedia.org/wiki/MapReduce) across hundreds or thousands of instances!

HOFs FTW!
Haskell Crash Course Part III

Writing Applications

Lets write the classic “Hello world!” program.

For example, in Python you may write:

```python
def main():
    print "hello, world!"
main()
```

and then you can run it:

```
$ python hello.py
hello world!
```
Haskell is a Pure language.

Not a value judgment, but a precise technical statement:

The “Immutability Principle”:

- A function must always return the same output for a given input
- A function’s behavior should never change

\[ \text{foo} : \text{In} \rightarrow \text{Out} \]

No Side Effects

Haskell’s most radical idea: expression \( \Rightarrow \) value

- When you evaluate an expression you get a value and
- Nothing else happens  "pure"
Specifically, evaluation must not have an side effects

- change a global variable or
- print to screen or
- read a file or
- send an email or
- launch a missile.

But… how to write “Hello, world!”

But, we want to …

- print to screen
- read a file
- send an email

Thankfully, you can do all the above via a very clever idea: Recipe
Recipes

This analogy is due to Joachim Brietner (https://www.seas.upenn.edu/~cis194/fall16/lectures/06-io-and-monads.html)

Haskell has a special type called IO – which you can think of as Recipe

```
type Recipe a = IO a
```

A value of type Recipe a

- is a description of a computation that can have side-effects
- which when executed performs some effectful I/O operations
- to produce a value of type a.

Recipes have No Side Effects

A value of type Recipe a is
• A description of a computation that can have side-effects

Cake vs. Recipe

(L) chocolate cake, (R) a sequence of instructions on how to make a cake.

They are different (Hint: only one of them is delicious.)

Merely having a Recipe Cake has no effects! The recipe

• Does not make your oven hot
• Does not make your floor dirty

Only One Way to Execute Recipes

Haskell looks for a special value

main :: Recipe ()
The value associated with `main` is handed to the runtime system and executed.

Baker Aker

The Haskell runtime is a *master chef* who is the only one allowed to cook!

---

**How to write an App in Haskell**

Make a Recipe () that is handed off to the master chef `main`.

- `main` can be arbitrarily complicated
- composed of smaller sub-recipes
A Recipe to Print to Screen

putStrLn :: String -> Recipe()

The function putStrLn

- takes as input a String
- returns as output a Recipe()

putStrLn msg is a Recipe() - when executed prints out msg on the screen.

main :: Recipe()
main = putStrLn "Hello, world!"

... and we can compile and run it

$ ghc --make hello.hs
$ ./hello
Hello, world!
QUIZ: How to Print Multiple Things?

Suppose I want to print two things e.g.

```
$ ghc --make hello.hs
$ ./hello2
Hello!
World!
```

Can we try to compile and run this:

```
main = (putStrLn "Hello!", putStrLn "World!")
```

A. Yes!

B. No, there is a type error!

C. No, it compiles but produces a different result!
A Collection of Recipes

Is just ... a collection of Recipes!

recPair :: (Recipe (), Recipe ())
recPair = (putStrLn "Hello!", putStrLn "World!")

recList :: [Recipe ()]
recList = [putStrLn "Hello!", putStrLn "World!"]

... we need a way to combine recipes!

Combining? Just do it!

We can combine many recipes into a single one using a do block

foo :: Recipe a3
foo = do r1 -- r1 :: Recipe a1
        r2 -- r2 :: Recipe a2
        r3 -- r3 :: Recipe a3

(or if you prefer curly braces to indentation)
foo = do { r1; -- r1 :: Recipe a1
          r2; -- r2 :: Recipe a2
          r3 -- r3 :: Recipe a3 }

The do block combines sub-recipes \( r_1 \), \( r_2 \) and \( r_3 \) into a new recipe that

- Will execute each sub-recipe in sequence and
- Return the value of type \( a_3 \) produced by the last recipe \( r_3 \)

Combining? Just do it!

So we can write

```haskell
main = do putStrLn "Hello!"
       putStrLn "World!"
```

or if you prefer

```haskell
main = do { putStrLn "Hello!"
           putStrLn "World!" }
```
EXERCISE: Combining Many Recipes

Write a function called `sequence` that

- Takes a list of recipes \([r_1, \ldots, r_n]\) as input and
- Returns a single recipe equivalent to `do \{r_1; \ldots; r_n\}

```
sequence :: [Recipe a] -> Recipe a
sequence rs = ???
```

When you are done you should see the following behavior

```-- Hello.hs
main = sequence [putStrLn "Hello!", putStrLn "World!"]
```

and then

```
$ ghc --make Hello.hs
$ ./hello
Hello!
World!
```
Using the Results of (Sub-) Recipes

Suppose we want a function that asks for the user’s name

```
$ ./hello
What is your name?
Ranjit
Hello Ranjit!
```

We can use the following sub-recipes

```
-- | read and return a line from stdin as String
getLine :: Recipe String

-- take a string s, return a recipe that prints s
putStrLn :: String -> Recipe ()
```

But how to

- Combine the two sub-recipes while
- Passing the result of the first sub-recipe to the second.
Naming Recipe Results via “Assignment”

You can write

\[ x \leftarrow \text{recipe} \]

to name the result of executing \text{recipe}

- \( x \) can be used to refer to the result in \text{later code}

Naming Recipe Results via “Assignment”

Let's write a function that asks for the user’s name

main = ask

ask :: Recipe ()
ask = do name <- getLine;
       putStrLn ("Hello " ++ name ++ "!")
Which produces the desired result

```
$ ./hello
What is your name?
Ranjit        # user enters
Hello Ranjit!
```

**EXERCISE**

Modify the above code so that the program *repeatedly* asks for the users’s name *until* they provide a *non-empty* string.

```haskell
-- Hello.hs

main = repeatAsk

repeatAsk :: Recipe ()
repeatAsk = _fill_this_in

isEmpty :: String -> Bool
isEmpty s = length s == 0
```
When you are done you should get the following behavior

$ ghc --make hello.hs

$ ./hello
What is your name?
# user hits return
What is your name?
# user hits return
What is your name?
# user hits return
What is your name?
Ranjit # user enters
Hello Ranjit!

**EXERCISE**

Modify your code to also print out a count in the prompt
$ ghc --make hello.hs

$ ./hello

(0) What is your name? # user hits return
(1) What is your name? # user hits return
(2) What is your name? # user hits return
(3) What is your name? # user enters
Ranjit # user enters
Hello Ranjit!

That’s all about IO

You should be able to implement build from Directory.hs

Using these library functions imported at the top of the file

```
import System.FilePath  (takeDirectory, takeFileName, (</>))
import System.Directory  (doesFileExist, listDirectory)
```

The functions are

- takeDirectory
- takeFileName
- (</>)
- doesFileExist
- listDirectory

hoogle the documentation to learn about how to use them.