The Haskell HOP: Higher-order Programming

COS 441 Slides 6

Slide content credits: Ranjit Jhala, UCSD

Agenda

- Haskell so far:
 - First-order functions
- This time:
 - Higher-order functions:
 - Functions as data, arguments & results
 - Reuseable abstractions
 - Capturing recursion patterns
 - Functional programming really starts to differentiate itself!

FUNCTIONS AS FIRST CLASS VALUES

A Perspective on Java

- In Java, you can do lots of things with integers:
 - create them whereever you want, in any bit of code
 - operate on them (add, subtract, etc)
 - pass them to functions, return them as results from functions
 - store them in data structures
- In Java, you can do barely anything at all with a method:
 - all you can do is declare a method inside a pre-existing class
 - you can't pass them to functions
 - you can't return them as results
 - you can't store them in data structures
 - you can't define them locally where you need them
 - of course, you can declare an entire new class (at the top level) and put the one method you are interested in inside it
 - this is incredibly heavy weight and still isn't very flexible!!
 - you still can't define methods locally where you want them

Functions as First-Class Data

- Haskell treats functions as first-class data. So does:
 SML, OCaml, Scala (an OO language)
- "First-class" == all the "privileges" of any other data type:
 - you can declare them where ever you want
 - declarations can depend upon local variables in the context
 - you can pass them as arguments to functions
 - you can return them as results
 - you can store them in data structures
- This feature makes it easy to create powerful abstractions
- Because it is easy, it encourages a programming style in which there is great code reuse, many abstractions and clear code

Functions as First-Class Data

• An example:

```
plus1 x = x + 1
minus1 x = x - 1
```

• Storing functions in data structures:

funp :: (Int -> Int, Int -> Int)
funp = (plus1, minus1)

• .. any data structure:

funs :: [Int -> Int]
funs = [plus1, minus1, plus1]

• An example:

doTwice f x = f (f x)

• Using it:

plus2 :: Int -> Int plus2 = doTwice plus1

• An example:

doTwice f x = f (f x)

• Using it:

plus2 :: Int -> Int plus2 = doTwice plus1

• Reasoning about it:

plus2 3

• An example:

doTwice f x = f (f x)

• Using it:

plus2 :: Int -> Int plus2 = doTwice plus1

• Reasoning about it:

plus2 3 = (doTwice plus1) 3

(unfold plus2)

• An example:

doTwice f x = f (f x)

• Using it:

plus2 :: Int -> Int plus2 = doTwice plus1

• Reasoning about it:

plus2 3 = (doTwice plus1) 3 = doTwice plus1 3

(unfold plus2) (parenthesis convention)

(f x) y == f x y

• An example:

doTwice f x = f (f x)

• Using it:

plus2 :: Int -> Int plus2 = doTwice plus1

• Reasoning about it:

plus2 3 = (doTwice plus1) 3 = doTwice plus1 3 = plus1 (plus1 3) = plus1 (3 + 1) = plus1 4 = 4 + 1 = 5

(unfold plus2)
(parenthesis convention)
(unfold doTwice)
(unfold plus1)
(def of +)
(unfold plus1, def of +)

(f x) y == f x y

Interlude

• What have we learned?

Interlude

- What have we learned? Almost nothing!
 - function application is left-associative:
 - ((f x) y) z == f x y z
 - like + or is left-associative:
 - (3 4) 6 == 3 4 6
 - this is useful, but intellectually uninteresting
- We have, however, unlearned something important:
 - some things one might have thought were fundamental differences between functions and other data types, turn out not to be differences at all!
- PL researchers (like me!) often work with the theory of functional languages because they are uniform and elegant
 - they don't make unnecessary distinctions
 - they get right down to the essentials, the heart of computation
 - at the same time, they do not lack expressiveness

Functions as Results

 Rather than writing multiple functions "plus1", "plus2", "plus3" we can write one:

```
plusn :: Int -> (Int -> Int)
plusn n = f
where f x = x + n
```

- plusn returns a function -- one that adds n to its argument
- any time we need an instance of plus, it is easy to build one:

plus10 :: Int -> Int plus10 = plusn 10

• we can also use plusn directly:

result1 = (plusn 25) 100

Functions as Results

• More trivial reasoning:

```
result1 = (plusn 25) 100
= (f) 100 where f x = x + 25
= 100 + 25
= 125
```

(unfold plusn) (unfold f) (def of +)

> plusn :: Int -> (Int -> Int) plusn n = f where f x = x + n

Precedence & Partial Application

• Function app is left-assoc.; Function types are right-assoc.

(plusn 25) 100 == plusn 25 100

Int -> (Int -> Int) == Int -> Int -> Int

• We've seen two uses of plusn:

plus20 = plusn 20

partial application

oneTwentyFive = plusn 25 100

- Whenever we have a function f with type T1 -> T2 -> T3, we can choose:
 - apply f to both arguments right now, giving a T3
 - partially applying f, ie: applying f to one argument, yielding new function with type T2 -> T3 and a chance to apply the new function to a second argument later

Defining higher-order functions

• The following was a stupid way to define plusn --- but it made it clear plusn was indeed returning a function:

```
plusn :: Int -> Int -> Int
plusn n = f
where f x = x + n
```

This is more beautiful code:

plusn' :: Int -> Int -> Int plusn' n x = x + n

• We can prove them equivalent for all arguments a and b

```
plusn a b = f b where f x = x + a(unfold plusn)= b + a(unfold f)= plusn' a b(fold plusn')
```

• So of course we can partially apply plusn' just like plusn

ANONYMOUS FUNCTIONS

Anonymous Numbers

You are all used to writing down numbers inside expressions
 This:

2 + 3

Is way more compact than this:

two = 2 three = 3 sum = two + three

– Why can't functions play by the same rules?

Anonymous Numbers

• Compare:

plus1 x = x + 1minus1 x = x - 1doTwice f x = f (f x)

baz = doTwice plus1 3
bar = doTwice minus1 7

```
doTwice f x = f (f x)
```

```
baz' = doTwice (\x -> x + 1) 3
bar' = doTwice (\x -> x - 1) 7
```

• When are anonymous functions a good idea?

function with argument x

- When functions are small and not reused.
- Why is this a good language feature?
 - It encourages the definition of abstractions like doTwice
 - Why? Without anonymous functions, doTwice would be a little harder to use -- heavier weight; programmers would do it less
 - Moreover, why make different rules for numbers vs. functions?

• Do you like shell scripting? Why not build your own pipeline operator in Haskell?

(|>) x f = f x

define an infix operator by putting a name made of symbols inside parens

arguments, body are the same as usual

• Use it:

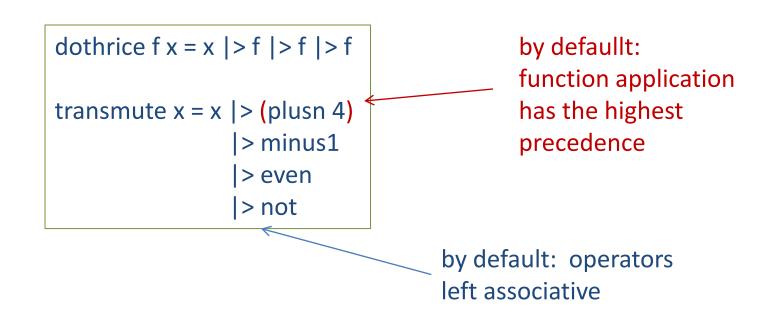
• Do you like shell scripting? Why not build your own pipeline operator in Haskell?

(|>) x f = f x

define an infix operator by putting a name made of symbols inside parens

arguments, body are the same as usual

• Use it:



• Do you like shell scripting? Why not build your own pipeline operator in Haskell?

(|>) x f = f x

define an infix operator by putting a name made of symbols inside parens

arguments, body are the same as usual

• Use it:

- Understanding functions in Haskell often boils down to understanding their type
- What type does the pipeline operator have?

(|>) x f = f x

(|>) :: a -> (a -> b) -> b

- Read it like this: "for all types a and all types b, |> takes a value of type a and a function from a to b and returns a b"
- Hence:

(3 |> plus1) :: Int (a was Int, b was Int)
(3 |> even) :: Bool (a was Int, b was Bool)
("hello" |> putStrLn) :: IO () (a was String, b was IO ())

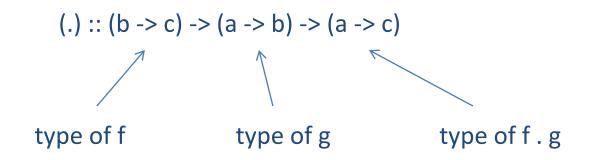
• Another heavily-used operator, function composition:

(.) fg x = f(g x)

• Another heavily-used operator, function composition:

(.) fg x = f(g x)

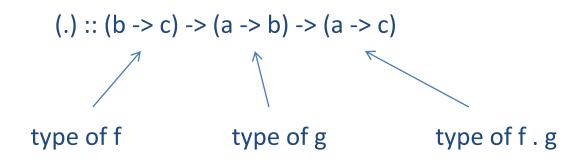
• What type does it have?



• Another heavily-used operator, function composition:

(.) fg x = f(g x)

• What type does it have?



• Examples:

plus2 = plus1. plus1

```
odd = even . plus1
```

bof = doTwice plus1 . doTwice minus1 baz = doTwice (plus1 . minus1) Exercise: prove equivalence

ABSTRACTING RECURSION PATTERNS

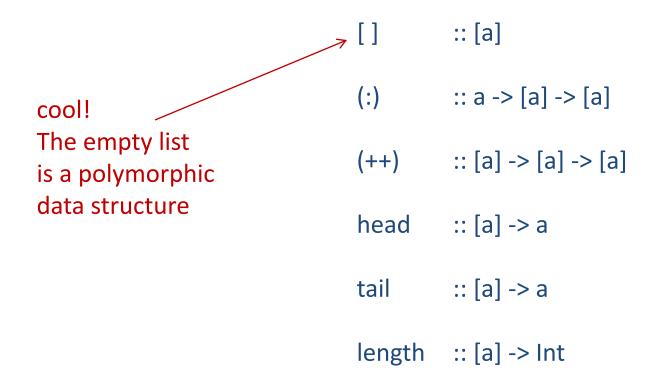
Abstracting Computation Patterns

- Higher-order functions and polymorphism are the "secret sauce" that really makes functional programming fun
- They make it not only possible but easy and delightful* for programmers to factor out repeated patterns in their code into highly reuseable routines
- It's especially effective in recursive routines -- one can sometimes eliminate the explicit recursion to be left with simple, non-recursive and abundantly clear code.

* Some people find delight from different sources than I do.

Recall: Polymorphic Lists

- Lists are heavily used in Haskell and other functional programming languages because they are light-weight, built-in "collection" data structure
- However, every major idea we present using lists applies similarly to any collection data structure we might define
- Recall some of the basic operations:



• Recall that strings are lists:

type String = [Char]

• Suppose we want to convert all characters to upper case:

toUpperString :: String -> String toUpperString [] = [] toUpperString (x:xs) = toUpper x : toUpperString xs

• Here I've applied to Upper to all elements of the list

Comment: try finding functions like "toUpper" by searching by type on http://haskell.org/hoogle

• Similar idioms come up often, even in completely different applications:

type Point= (Int, Int) type Vector = (Int, Int) type Polygon = [XY]

• It is easy to move a single point:

shiftPoint :: Vector -> Point -> Point
shiftPoint (dx, dy) (x, y) = (x + dx, y + dy)

• And with more work, entire polygon:

```
shift :: Vector -> Polygon -> Polygon
shift d [ ] = [ ]
shift d (x:xs) = shiftPoint d x : shift d xs
```

• How to extract the pattern?

```
shift :: Vector -> Polygon -> Polygon
shift d [ ] = [ ]
shift d (x:xs) = shiftPoint d x : shift d xs
```

• VS

```
toUpperString :: String -> String
toUpperString [] = []
toUpperString (x:xs) = toUpper x : toUpperString xs
```

• How to extract the pattern?

```
shift :: Vector -> Polygon -> Polygon
shift d [ ] = [ ]
shift d (x:xs) = shiftPoint d x : shift d xs
```

```
• VS
```

```
toUpperString :: String -> String
toUpperString [] = []
toUpperString (x:xs) = toUpper x : toUpperString xs
```

• Here's the common pattern:

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

• map applies f to all elements of the list in place

• Rewriting:

toUpperString s = map toUpper s

and

shift d polygon = map (shiftPoint d) polygon

- Now that's delightful!
- Compare:

partial application

```
toUpperString [] = []
toUpperString (x:xs) = toUpper x : toUpperString xs
```

```
shift d [ ] = [ ]
shift d (x:xs) = shiftPoint d x : shift d xs
```

A step further

• Rewrite this:

```
toUpperString s = map toUpper s
```

```
shift d polygon = map (shiftPoint d) polygon
```

• To this:

toUpperString = map toUpper shift d = map (shiftPoint d)

A step further

• Rewrite this:

```
toUpperString s = map toUpper s
```

shift d polygon = map (shiftPoint d) polygon

• To this:

toUpperString = map toUpper shift d = map (shiftPoint d)

• In general, rewrite:

f x = e x

• To

f = e (when x does not appear in e)

this is quite common but I actually find it harder to read

the syntactic redundancy with argument "x" gives me a hint about the type

• Two more functions:

```
listAdd [ ] = 0
listAdd (x:xs) = x + (listAdd xs)
listMul [ ] = 1
listMul (x:xs) = x * (listMul xs)
```

• You can see the syntactic pattern. How do I capture it?

• Two more functions:

```
listAdd [ ] = 0
listAdd (x:xs) = x + (listAdd xs)
listMul [ ] = 1
listMul (x:xs) = x * (listMul xs)
```

• You can see the syntactic pattern. How do I capture it?

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

• Two more functions:

```
listAdd [] = 0
listAdd (x:xs) = x + (listAdd xs)
listMul [] = 1
listMul (x:xs) = x * (listMul xs)
```

• You can see the syntactic pattern. How do I capture it?

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

```
listAdd = foldr 0 (+)
```

```
listMul = foldr 1 (*)
```

• Some more folds:

length [] = 0length (x:xs) = 1 + (length xs) length xs =

factorial 0 = 1 factorial n = n * (factorial (n-1)) factorial n =

sequence_ :: [IO ()] -> IO () sequence as =
sequence_ [] = null
sequence_ (a:as) = a >> sequence_ as

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

• Some more folds:

length [] = 0length (x:xs) = 1 + (length xs) length xs = foldr 0 (1+) xs

factorial 0 = 1 factorial n = n * (factorial (n-1)) factorial n =

sequence_ :: [IO ()] -> IO () sequence as =
sequence_ [] = null
sequence_ (a:as) = a >> sequence_ as

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

• Some more folds:

length [] = 0length (x:xs) = 1 + (length xs) length xs = foldr 0 (1+) xs

factorial 0 = 1 factorial n = n * (factorial (n-1)) factorial n = foldr 1 (*) [1..n]

sequence_ :: [IO ()] -> IO () sequence as =
sequence_ [] = null
sequence_ (a:as) = a >> sequence_ as

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

• Some more folds:

length [] = 0length (x:xs) = 1 + (length xs) length xs = foldr 0 (1+) xs

factorial 0 = 1 factorial n = n * (factorial (n-1)) factorial n = foldr 1 (*) [1..n]

sequence_ :: [IO ()] -> IO ()
sequence_ [] = null
sequence_ (a:as) = a >> sequence_ as

sequence as = foldr null (>>) as

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

map :: (a -> b) -> [a] -> [b]

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

• Can we define map in terms of foldr?

map :: (a -> b) -> [a] -> [b]

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

• Can we define map in terms of foldr?

map f xs = foldr [] (\x ys -> f x : ys) xs

map :: (a -> b) -> [a] -> [b]

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

• Can we define foldr in terms of map?

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

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

- Can we define foldr in terms of map?
 - No. How do we prove it?
 - A formal theorem might say:
 - for all b, f, xs, there exists g, ys such that foldr b f xs == map g ys

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

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

- Can we define foldr in terms of map?
 - No. How do we prove it?
 - A formal theorem might say:
 - for all b, f, xs, there exists g, ys such that foldr b f xs == map g ys
 - To disprove that theorem, find a counter-example. Consider:
 - length xs = foldr 0 (1+) xs
 - Does there exist a g and ys such that
 - fold 0 (1+) xs == map g ys ?

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

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

- Can we define foldr in terms of map?
 - No. How do we prove it?
 - A formal theorem might say:
 - for all b, f, xs, there exists g, ys such that foldr b f xs == map g ys
 - To disprove that theorem, find a counter-example. Consider:
 - length xs = foldr 0 (1+) xs
 - Does there exist a g and ys such that
 - fold 0 (1+) xs == map g ys ?
 - Consider the types:
 - fold 0 (1+) xs :: Int

incomparable types no matter what b is!

• map g ys :: [b] 4

Exercises

- Lists are one kind of container data structure; they support
 - map: the "apply all in place" pattern
 - fold: "the accumulative iteration" pattern
- What about trees?

data Tree a = Leaf a | Branch (Tree a) (Tree a)

- Define treeMap and treeFold
- Give them appropriate types
- Can you define treeMap in terms of treeFold? Vice versa?

A NOTE ON I/O

A Note on I/O

• What is the null action?

```
null :: IO ()
null = return ()
```

- return is very (very!) different from return in Java or C
- "return v" creates an action that has no effect but results in v
 return "hi" -- action that returns the string "hi" and does nothing else
 return () -- action that returns the unit value () and does nothing else

A Note on I/O

- We can use return in conjunction with do notation
- Example:

do		do	
-	<- return "hi" outStrLn s	=	putStrLn "hi"
In general:			
do			do
Х	<- return e		e e
	. x x		

- This is another powerful law for reasoning about programs using substitution of equals for equals
- The fascinating thing is that it interacts safely with effects
- More on this later!

SUMMARY

Summary

- Higher-order programs
 - receive functions as arguments
 - return functions as results
 - store functions in data structures
 - use anonymous functions wisely
- Great programmers identify repeated patterns in their code and devise higher-order functions to capture them
 - map and fold are two of the most useful