Session 7: Recursion and Higher-Order Functions

COMP2221: Functional programming

Laura Morgenstern

laura.morgenstern@durham.ac.uk

Recap

- Contrasted sum and product types, and availability in other languages
- Discussed the pros and cons of classes and algebraic data types
- Considered how to write recursive functions
- Classified recursive functions: linear vs. multi recursion, direct vs. indirect recursion

Recursion Continued

Complexity

Is this a good implementation?

- The reverse of a list is computed by appending the head onto the reverse of the tail.

```
reverse' :: [a] -> [a]
reverse' []     = []
reverse' (x:xs) = reverse' xs ++ [x]
```

Complexity

Is this a good implementation?

- The reverse of a list is computed by appending the head onto the reverse of the tail.

```
reverse' :: [a] -> [a]
reverse' []     = []
reverse' (x:xs) = reverse' xs ++ [x]
```

```
reverse' [1, 2, 3]
== reverse' [2, 3] ++ [1]                 -- applying reverse'
== (reverse' [3] ++ [2]) ++ [1]           -- applying reverse'
== ((reverse' [] ++ [3]) ++ [2]) ++ [1]  -- base case
== (([] ++ [3]) ++ [2]) ++ [1]            -- applying (++)
== ([3] ++ [2]) ++ [1]                    -- applying (++)
== [3, 2] ++ [1]                          -- applying (++)
== [3, 2, 1]
```

Complexity

Is this a good implementation?

- The reverse of a list is computed by appending the head onto the reverse of the tail.

```
reverse' :: [a] -> [a]
reverse' []     = []
reverse' (x:xs) = reverse' xs ++ [x]
```

```
reverse' [1, 2, 3]
== reverse' [2, 3] ++ [1]                -- applying reverse'
== (reverse' [3] ++ [2]) ++ [1]          -- applying reverse'
== ((reverse' [] ++ [3]) ++ [2]) ++ [1] -- base case
== (([] ++ [3]) ++ [2]) ++ [1]           -- applying (++)
== ([3] ++ [2]) ++ [1]                   -- applying (++)
== [3, 2] ++ [1]                         -- applying (++)
== [3, 2, 1]
```

- Recall that `(++)` must *traverse* its first argument
- So this implementation is $\mathcal{O}(n^2)$ in the length of the input list

Complexity

A more efficient way: combine reverse and append

```haskell
-- helper function
reverse'' :: [a] -> [a] -> [a]
reverse'' [] ys     = ys
reverse'' (x:xs) ys = reverse'' xs (x:ys)

reverse' :: [a] -> [a]
reverse' xs = reverse'' xs []
```

Complexity

A more efficient way: combine reverse and append

```haskell
-- helper function
reverse'' :: [a] -> [a] -> [a]
reverse'' [] ys     = ys
reverse'' (x:xs) ys = reverse'' xs (x:ys)

reverse' :: [a] -> [a]
reverse' xs = reverse'' xs []

reverse' [1, 2, 3, 4]
== reverse'' [1, 2, 3, 4] []  -- applying reverse'
== reverse'' [2, 3, 4] (1:[]) -- applying reverse''
== reverse'' [3, 4] (2:1:[])  -- applying reverse''
== reverse'' [4] (3:2:1:[])   -- applying reverse'
== reverse'' [] (4:3:2:1:[])  -- base case
== (4:3:2:1:[])               -- applying (:)
== [4, 3, 2, 1]
```

- Since `(:)` is $\mathcal{O}(1)$, this implementation is $\mathcal{O}(n)$.

Debugging errors

- Easy to get confused writing recursive functions
- The case enumeration is useful
- Helpful to write out the call stack "by hand" for a small example
- Usual error is that not all base cases are covered

Higher Order Functions

Higher order functions

- We've seen many functions that are naturally recursive
- We'll now look at *higher order functions* in the standard library that capture many of these patterns

Definition (Higher order function)

A function that does at least one of

- take one or more functions as arguments
- returns a function as its result

- Due to currying, every function of more than one argument is higher-order in Haskell

```haskell
add :: Num a => a -> a -> a
add x y = x + y

-- "add 1" returns a function!
Prelude> :type add 1
Num a => a -> a
```

Examples for higher order functions on lists

- Many *linear recursive* functions on lists can be written using higher order library functions
- `map`: apply a function to all elements in a list

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

- `filter`: select elements from a list that satisfy a predicate

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p xs = [x | x <- xs, p x]
```

- `any`, `all`, `foldr`, `takeWhile`, `dropWhile`, ...
- For more, see `http://hackage.haskell.org/package/base-4.12.0.0/docs/Prelude.html#g:13`

Function composition

- Often tedious to write brackets and explicit variable names
- Can use *function composition* to simplify this

$$(f \circ g)(x) = f(g(x))$$

- Haskell uses the `(.)` operator

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \x -> f (g x)
-- example
odd a = not (even a)
odd   = not . even -- Point-free style: no need for the variable a
```

- Useful for writing composition of functions to be passed to other higher order functions
- Removes need to write λ-expressions
- Called "pointfree" style.

- Saw example higher-order functions on lists
- Now we'll get even more generic and implement these generic patterns for custom datatypes

Functors

Use type classes to implement generic higher order functions

- Recall, Haskell has a concept of *type classes*
- These describe interfaces that can be used to constrain the polymorphism of functions to those types satisfying the interface

Example

- `(+)` acts on any type, as long as that type implements the `Num` interface

```
(+) :: Num a => a -> a -> a
```

- `(<)` acts on any type, as long as that type implements the `Ord` interface

```
(<) :: Ord a => a -> a -> Bool
```

- Haskell comes with *many* such type classes encapsulating common patterns
- When we implement our own data types, we can "just" implement appropriate instances of these classes

Use type classes to implement generic higher order functions

- Recall, Haskell has a concept of *type classes*
- These describe interfaces that can be used to constrain the polymorphism of functions to those types satisfying the interface
- Haskell has *many* type classes encapsulating common patternsin the standard library:
  - `Num`: numeric types
  - `Eq`: equality types
  - `Ord`: orderable types
  - `Functor`: mappable types
  - `Foldable`: foldable types
  - ...
- If you implement a new data type, it is worthwhile thinking if it satisfies any of these interfaces
- When we implement our own data types, we can "just" implement appropriate instances of these classes

Let's look at the types of two "map" functions

```haskell
data [] a = [] | a:[a]
map :: (a -> b) -> [a] -> [b]

data BinaryTree a = Leaf a | Node a (BinaryTree a) (BinaryTree a)
bmap :: (a -> b) -> BinaryTree a -> BinaryTree b
```

Only difference is the type name of the container. This suggests that we should make a "Container" type class to capture this pattern.

Haskell calls this type class `Functor`

```haskell
class Functor c where
  fmap :: (a -> b) -> c a -> c b
```

If a type implements the `Functor` interface, it defines a data structure that we can transform the elements of in a systematic way.

`fmap`: a generic map function

```
Prelude> :t fmap
fmap :: Functor f => (a -> b) -> f a -> f b
Prelude> fmap (*2) [1, 2, 3]
[2, 4, 6]

class Functor f where
   fmap :: (a -> b) -> f a -> f b
```

- Works on any mappable structure
- Must obey *functor laws*:
- `fmap id c == c` Mapping the identity function over a structure should return the structure untouched.
- `fmap f (fmap g c) == fmap (f . g) c` Mapping over a container should distribute over function composition (since the structure is unchanged, it shouldn't matter whether we do this in two passes or one).

Instance declaration for Functors

Use an *instance* declaration to attach an `fmap` implementation to a container type.

```
data List a = Nil | Cons a (List a)
  deriving (Eq, Show)

instance Functor List where
  fmap _ Nil = Nil
  fmap f (Cons a tail) = Cons (f a) (fmap f tail)

data BinaryTree a = Leaf a | Node a (BinaryTree a) (BinaryTree a)
  deriving (Eq, Show)

instance Functor BinaryTree where
  fmap f (Leaf a) = Leaf (f a)
  fmap f (Node a l r) = Node (f a) (fmap f l) (fmap f r)
```

Generic code

```haskell
list = Cons 1 (Cons 2 (Cons 4 Nil))
btree = Node 1 (Leaf 2) (Leaf 4)

-- Generic add1
add1 :: (Functor c, Num a) => c a -> c a
add1 = fmap (+1)

Prelude> add1 list
Cons 2 (Cons 3 (Cons 5 Nil))
Prelude> add1 btree
Node 2 (Leaf 3) (Leaf 5)
```

Correctness of `listMap`

```haskell
data List a = Nil | Cons a (List a) deriving (Eq, Show)

instance Functor List where
  fmap _ Nil = Nil
  fmap f (Cons x xs) = Cons (f x) (fmap f xs)
```

To show `fmap id == id`, need to show `fmap id (Cons x xs) == Cons x xs` for any x, xs.

```haskell
-- Induction hypothesis
fmap id xs = xs
-- Base case
-- apply definition
fmap id Nil = Nil
-- Inductive case
fmap id (Cons x xs) = Cons (id x) (fmap id xs)
== Cons x (fmap id xs)
== Cons x xs -- Done!
```

Exercise: check whether the second law holds

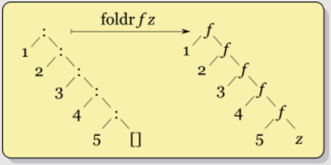
Folds: a family of higher order functions

Folds

- *folds* process a data structure in some order and build a return value
- Haskell provides a number of these in the standard prelude, with more available in the `Data.List` module

`foldr`: right associative fold

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

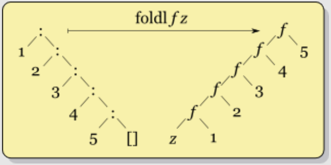


Folds

- *folds* process a data structure in some order and build a return value
- Haskell provides a number of these in the standard prelude, with more available in the `Data.List` module

`foldl`: left associative fold

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f z []     = z
foldl f z (x:xs) = foldl f (z `f` x) xs -- tail recursive!
```



How to think about this

- `foldr` and `foldl` are recursive
- However, often easier to think of them *non-recursively*

foldr

Replace `(:)` by the given function, and `[]` by given value.

```
sum [1, 2, 3]
= foldr (+) 0 [1, 2, 3]
= foldr (+) 0 (1:(2:(3:[])))
= 1 + (2 + (3 + 0))
= 6
```

foldl

Same idea, but associating to the left

```
sum [1, 2, 3]
= foldl (+) 0 [1, 2, 3]
= foldl (+) 0 (1:(2:(3:[])))
= (((1 + 2) + 3) + 0)
= 6
```

Purpose of folds

- Capture many linear recursive patterns in a clean way
- Can have efficient library implementation $\Rightarrow$ can apply program optimisations
- Actually apply to all `Foldable` types, not just lists
- e.g. `foldr`'s type is actually

```haskell
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
```

- So we can write code for lists and (say) trees identically

Folds are general

- Many library functions on lists are written *using folds*

```haskell
product = foldr (*) 1
sum = foldr (+) 0
maximum = foldr1 max
```

- Practical sheet 4 asks you to define some others

Which to choose?

foldr

- Generally `foldr` is the right choice
- Works even for infinite lists
- Note `foldr (:) [] == id`
- Can terminate early

foldl

- Usually best to use *strict* version:

```haskell
import Data.List
foldl' -- note trailing '
```

- Doesn't work on infinite lists (needs to start at the end)
- Use when you *want* to reverse the list: `foldl (flip (:)) [] == reverse`
- Can't terminate early

⇒ CAUTION: `foldr` and `foldl` lead to different result if operator `f` not commutative

Foldable data structures

- `Foldable` type class: if we can *combine* an `a` and a `b` to produce a new `b`, then, given a start value and a container of `a`s I can turn it into a `b`

```
class Foldable f where
  -- minimal definition requires this
  foldr :: (a -> b -> b) -> b -> f a -> b
```

Summary

- Introduced definition of *higher order functions*
- Saw definition and use of a number of such functions on lists
- Talked about *Foldable* and *Functor* to capture generic *patterns* of computation