Session 5: Recursion and higher order functions

COMP2221: Functional programming

Lawrence Mitchell*

*lawrence.mitchell@durham.ac.uk

Recap

- Saw nameless or anonymous functions (λ-expressions), and syntax

```
length' :: [a] -> Int
length' xs = sum (map (\_ -> 1) xs)
```

- Discussed why you might want to use them: only use the function once, clarity, thinking about function composition.
- Discussed a little *implementation* of lists: linked lists
- Talked about the list constructor *cons* `(:)`
- Saw how pattern matching can be used to match lists, and discussed wildcard patterns

```
takeFirstTwo :: [a] => (a, a)
takeFirstTwo (x:y:_) = (x, y)
```

Proof by induction
⇔ (correct) recursive functional implementation.

Definition

recursion *noun*

see: recursion.

program provides a witness to a proof of some statement.
↳ underpins "proof assistants" automated theorem provers.

Solving problems with recursion

Solve simple problems

- If the given instance of the problem can be solved directly, solve it directly.
- Otherwise, reduce it to one or more *simpler* instances of the same problem.

Enter ***the recursion fairy****:*
Your only task is to simplify the original problem, or to solve it directly when simplification is either unnecessary or impossible; the Recursion Fairy will solve all the simpler subproblems for you, using Methods That Are None Of Your Business So mind your own business.

Jeff Erickson, Algorithms

`https://jeffe.cs.illinois.edu/teaching/algorithms/`

Advice when writing recursive functions

Translate into Haskell.

1. define the type
2. enumerate the cases → write out patterns
3. define the simple or *base* cases ←
4. define the reduction of other cases to simpler ones ← problem ⇒ simpler problems
5. (optional) generalise and simplify

Example: `drop`

1. define the type

Drop the first *n* elements from a list

```
drop :: Int -> [a] -> [a]
```

2. enumerate the cases
3. define the simple or *base* cases
4. define the reduction of other cases to simpler ones
5. (optional) generalise and simplify

Example: `drop`

1. define the type

Drop the first *n* elements from a list

```
drop :: Int -> [a] -> [a]
```

2. enumerate the cases

Two cases each for the integer and the list argument

```
drop 0 [] =
drop 0 (x:xs) =
drop n [] =
drop n (x:xs) =
```

3. define the simple or *base* cases
4. define the reduction of other cases to simpler ones
5. (optional) generalise and simplify

Example: drop

1. define the type
2. enumerate the cases

Two cases each for the integer and the list argument

```
drop 0 [] =
drop 0 (x:xs) =
drop n [] =
drop n (x:xs) =
```

3. define the simple or *base* cases

Zero and the empty list are fixed points

```
drop 0 [] = []
drop 0 (x:xs) = x:xs
drop n [] = []
drop n (x:xs) =
```

4. define the reduction of other cases to simpler ones
5. (optional) generalise and simplify

Example: drop

1. define the type
2. enumerate the cases
3. define the simple or *base* cases

Zero and the empty list are fixed points

```
drop 0 [] = []
drop 0 (x:xs) = x:xs
drop n [] = []
drop n (x:xs) =
```

4. define the reduction of other cases to simpler ones

Apply drop to the tail

```
drop 0 [] = []
drop 0 (x:xs) = x:xs
drop n [] = []
drop n (x:xs) = drop (n-1) xs
```

5. (optional) generalise and simplify

Example: `drop`

1. define the type
2. enumerate the cases
3. define the simple or *base* cases
4. define the reduction of other cases to simpler ones

Apply drop to the tail

```
drop 0 [] = []
drop 0 (x:xs) = x:xs
drop n [] = []
drop n (x:xs) = drop (n-1) xs
```

5. (optional) generalise and simplify

Compress cases

```
drop :: Int -> [a] -> [a]
drop 0 xs = xs
drop _ [] = []
drop n (x:xs) = drop (n-1) xs
```

Example: `drop`

1. define the type
2. enumerate the cases
3. define the simple or *base* cases
4. define the reduction of other cases to simpler ones
5. (optional) generalise and simplify

Compress cases

```
drop :: Int -> [a] -> [a]
drop 0 xs = xs
drop _ [] = []
drop n (x:xs) = drop (n-1) xs
```

6. And we're done (this is the standard library definition)

Equivalence of recursion and iteration

- Both purely iterative and purely recursive programming languages are Turing complete
- Hence, it is always possible to transform from one representation to the other
- Which is convenient depends on the algorithm, and the programming language

Recursion ⇒ iteration

- Write looping constructs, manually manage function call stack

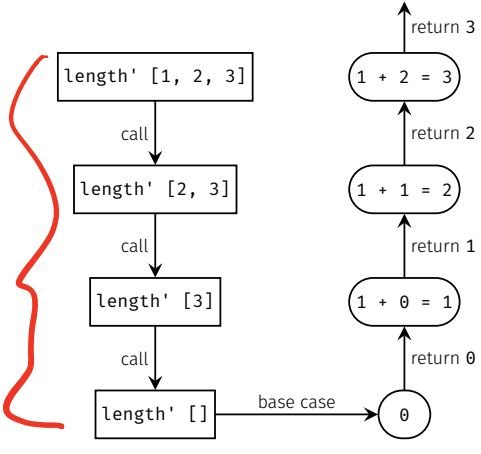
Iteration ⇒ recursion

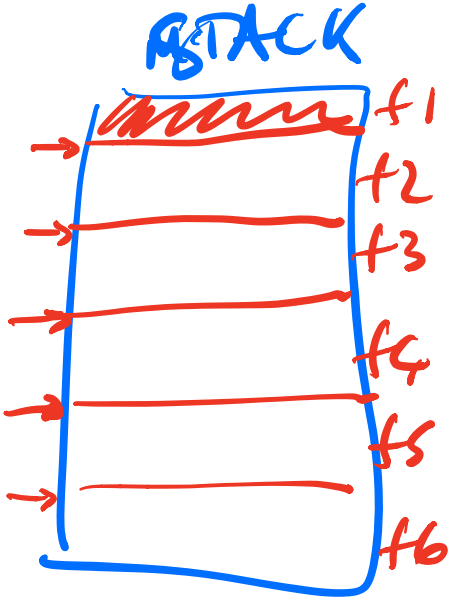
- Turn loop variables into additional function arguments
- and write a *tail recursive function* (see later)

How are function calls managed?

- Usually a *stack* is used to manage nested function calls

```
length' :: [a] -> Int
length' [] = 0
length' (x:xs) = 1 + length' xs
Prelude> length' [1, 2, 3]
```





- Each entry on the stack uses memory
- Too many entries causes errors: the dreaded *stack overflow*
- How big this stack is depends on the language
- Typically "small" in imperative languages and "big" in functional ones

Typically don't have to worry about stack overflows

- In traditional *imperative* languages, we often try and avoid recursion
- Function calls are more expensive than just looping
- Deep recursion can result in stack overflow:

```
def fac(n): return 1 if n == 0 else n * fac(n-1)
> fac(3000)
RecursionError Traceback (most recent call last)
----> 1 def fac(n): return 1 if n == 0 else n * fac(n-1)
RecursionError: maximum recursion depth exceeded in comparison
```

- In contrast, Haskell is fine with much deeper recursion

```
fac n = if n == 0 then 1 else n * fac (n-1)
> fac(200000)
..... -- fine, if slow
```

- Unsurprising, given the programming model
- Still prefer to avoid recursion trees that are too deep

Classifying recursive functions I

- Since it is natural to write recursive functions, it makes sense to think about classifying the different types we can encounter
- Classifying the type of recursion is useful to allow us to think about better/cheaper implementations

Definition (Linear recursion)

The recursive call contains only a *single* self reference

```
length' []     = []
length' (_:xs) = 1 + length' xs
```

Function just calls itself repeatedly until it hits the base case.

Definition (Multiple recursion)

The recursive call contains *multiple* self references

```
fib 0 = 0
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

Classifying recursive functions II

Definition (Direct recursion)

The function calls *itself* recursively

```
product' []     = []
product' (x:xs) = x * product' xs
```

Definition (Mutual/indirect recursion)

Multiple functions call *each other* recursively

```
even' :: Integral a => a -> Bool
even' 0 = True
even' n = odd' (n - 1)

odd' :: Integral a => a -> Bool
odd' 0 = False
odd' n = even' (n - 1)
```

Tail recursion: a special case

Definition (Tail recursion)

A function is *tail recursive* if the *last result of a recursive call* is the result of the function itself.

Loosely, the last thing a tail recursive function does is call itself with new arguments, or return a value.

- Such functions are useful because they have a trivial translation into loops
- Some languages (e.g. Scheme) *guarantee* that a tail recursive call will be transformed into a "loop-like" implementation using a technique called *tail call elimination*.

⇒ complexity remains unchanged, but implementation is more efficient.

- In Haskell implementations, while nice, this is not so important (other techniques are used)

Big here. Racket.

Iteration ⇔ tail recursion

Loops are convenient

```
def factorial(n):
    res = 1
    for i in range(n, 1, -1):
        res *= i
    return res
```

Tail recursive implementation

- We can't write this directly, since we're not allowed to mutate things
- We can write it with a helper recursive function where all loop variables become arguments to the function

```
factorial n = loop n 1
  where loop n res | n < 0      = undefined
                   | n > 1      = loop (n - 1) (res * n)
                   | otherwise = res
```

Examples

Not tail recursive

Calls `(*)` after recursing

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

Tail recursive

Recursive call to `loop` calls itself "outermost"

```
product' :: Num a => [a] -> a
product' xs = loop xs 1
  where loop [] n     = n
        loop (x:xs) n = loop xs (x * n)
```

Examples

Also for mutual recursion

Our `even`/`odd` functions are mutually tail recursive

```
even 0 = True
even n = odd (n-1)
odd 0 = False
odd n = even (n-1)

odd 4
==> even 3
==> odd 2
==> even 1
==> odd 0
==> False
```

What about complexity?

- Linear recursion often appears in list traversals. Typically make $\mathcal{O}(n)$ recursive calls on data of size n
- Multiple recursion often appears in tree or graph traversals, as well as "divide and conquer" algorithms (e.g. binary search). Number of recursive calls more problem dependent

Which would you use?

```haskell
binomial :: Integral a => a -> a -> a
binomial n 0 = 1
binomial n k
  | n == k    = 1
  | otherwise = binomial (n - 1) (k - 1) + binomial (n - 1) k

binomial' :: Integral a => a -> a -> a
binomial' n k = product [n-(k-1)..n] `div` product [1..k]
```

Careful of hidden costs I

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]
```

Careful of hidden costs I

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]
```

Careful of hidden costs I

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 $\mathcal{O}(n^2)$ is the length of the input list

Careful of hidden costs II

A more efficient way: *combine* reverse and append

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

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

Careful of hidden costs II

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 earlier advice is useful
- I often find it useful to write out the call stack “by hand” for a small example
- Usual error is that all base cases were not covered

Building block summary

- Prerequisites: none
- Content
  - Saw strategic approach for writing recursive functions
  - Briefly discussed theoretical *equivalence* of recursion and iteration
  - Saw how to *implement* "iteration" in recursive languages, and vice versa
  - Classified different types of recursive functions
  - Showed example of (hidden) pitfalls
- Expected learning outcomes
  - student can *implement* recursive functions
  - student can *explain* what class of recursion a function exhibits
  - student can *describe* how recursive calls can be implemented using stacks
- Self-study
  - None

Maps and folds

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

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

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

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

Why are they useful?

- *Common programming idioms* can be written as functions in the language
- *Domain specific languages* can be defined with appropriate collections of higher order functions
- We can use the *algebraic properties* of higher order functions to reason about programs $\Rightarrow$ provably correct *program transformations*

$\Rightarrow$ useful for domain specific *compilers* and automated program generation

Higher order functions on lists

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

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

- `filter`: remove entries from a list

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

- `any`, `all`, `concatMap`, `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 -- No need for the a variable
```

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

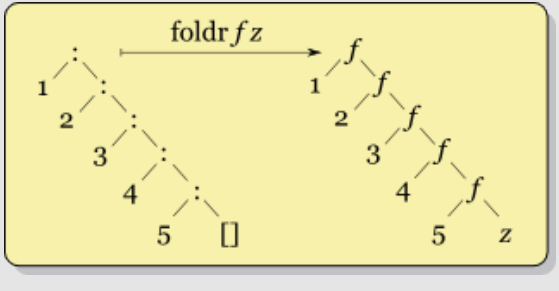
Folds

- *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

Processes list from the front

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



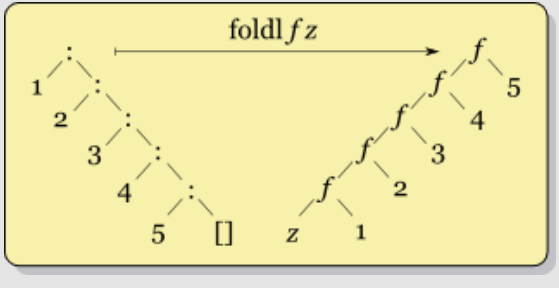
Folds

- *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

Processes list from the back (implicitly in reverse)

```
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
- 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) + 1)
= 6
```

Why would I use them?

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

```
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*

```
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 (ha!) choice
- Works even for infinite lists!
- Note `foldr (:) [] == id`
- Can terminate early.

foldl

- Usually best to use *strict* version:

```
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.

Building block summary

- Prerequisites: none
- Content
  - Introducted definition of *higher order functions*
  - Saw definition and use of a number of such functions on lists
  - Talked about *folds* and capturing a generic *pattern* of computation
  - Gave examples of why you would prefer them over explicit iteration
- Expected learning outcomes
  - student can *explain* what makes a function higher order
  - student can *write* higher order functions
  - student can *use* folds to realise linear recursive patterns
  - student can *explain* differences between `foldr` and `foldl`
- Self-study
  - None