Higher Order Functions, Partial Application

Confession

We lied!

- Functions in Haskell *cannot* take multiple arguments
- They can only take *one* argument
- What does a function type *really* mean?
  - `take :: Int -> [a] -> [a]`
  - If not two arguments, then what?
- How does `->` associate?
  - `Int -> [a] -> [a] == Int -> ([a] -> [a])`
  - `foldr :: (t1 -> t2 -> t2) -> t2 -> [t1] -> t2`
    - `(t1 -> (t2 -> t2)) -> (t2 -> ([t1] -> t2))`
  - It is *right* associative
- What does this really mean?

Only One Argument..

- `foldr :: (t1 -> t2 -> t2) -> t2 -> [t1] -> t2`
  - `(t1 -> (t2 -> t2)) -> (t2 -> ([t1] -> t2))`
- Take one argument and..
  - .. the type looks like a *function type*..
- Yes, the application of one argument returns a *function*!
- So `foldr (\x y->x+y) 0 [1..10]` is actually
  - `((foldr (\x y->x+y)) 0) [1..10]`
- We know what happens when we have all arguments, but when we do not?
- `drop :: Int -> ([a] -> [a])`
- `drop 2 :: [a] -> [a]`
  - a function taking a list and returns a list
  - a function that drops the first two elements
- When applying a function to “too few” arguments we get a new function
  - This is called *partial application*
- Functions with this behaviour are called *curried functions*
  - after Haskell B. Curry

Constructing Functions (pt 2)

- `(\x y .. -> …)` constructs an anonymous function
- Partial application can be used to construct new functions
  - new function is specialised, since first argument of original function is *fixed*
  - `take 3 :: [a] -> [a]`
    - function that returns the first three elements of a list
  - `map length :: [[a]] -> [Int]`
    - function computes the length of every element in a list
  - `(\n x -> mod x n) 7 :: Int -> Int`
    - function that returns its argument modulo 7
- Think about order of arguments when you define a function
  - Which arguments are most convenient to be part of a specialisation?
  - Then again, easy to fix with a lambda expression
    - `(\x y -> f y x)` like f but with swapped argument order
- Since functions are first class when can write functions that only take functions as arguments and make new functions.

Constructing Functions (pt 2)

- We can compose two functions so that the result of one application is fed to another.
- `compose f g x = f (g x)`
  - `compose :: (b->c) -> (a->b) -> a -> c`
  - Exists in Haskell as `(.)`
  - `(length . tail) “Haskell” == 6`
  - Note that `tail` is applied first and then `length` to the result
  - Background from math:
    - $(f \circ g)(x) = f(g(x))$, but sometimes defined as $(f \circ g)(x) = g(f(x))$
- `twice f = f . f`
  - `\x -> f (f x)`
- With the compose operator we *compactly* construct new functions without having to list arguments.
- Be careful: compact isn’t always the same as readable.

Constructing Functions (pt 2)

- We can also use partial application when defining named functions
- Partial application (fewer than all arguments) returns a function
- `inclist delta = map (\e-> e + delta)`
  - `map :: (a->b) -> [a] -> [b]`
  - `map (\e-> e + delta) :: [Int] -> [Int]`
  - `inclist :: Int -> [Int] -> [Int]`
  - `inclist 4 :: [Int] -> [Int]`
- We don't *need* to give all arguments due to partial application
  - if both sides end with the same sequence of arguments we can skip them
  - Note: it may not always be easy to read, though
- The combination of lazy evaluation, curried functions and partial application is taking some old ideas to an extreme.
  - Compact programs
  - Other functional languages have the possibility to create functions like this, but few have the same ease of partial application

Haskell shorthands

- When using a prefix function, i.e., `foo x y x`, partial application is easy, but what about all the infix functions in Haskell?
  - Haskell provides..
- We know that an infix operator `op` can be written `(op)`
- `x op y == (op) x y`
- We *can* do partial application with `(op) x`
- Even more convenient is to include one argument *within* the parentheses
  - `(+2) == \x -> x + 2`
  - `(2+) == \x -> 2 + x`
  - `('c':) == \x -> 'c':x`
  - `(:"foo") == \h -> h:"foo"`
  - `(!!10)` return 10th element of list
- This can be done for *all* infix operators
- We can convert any function to an infix operator with `` `..` ``
  - ``(`map` [1..10])``
  - ``(`foldr` 0)``

Abstraction (part 3)

- Curried functions, partial application and function composition provide additional steps to reduce clutter
  - Again, hide details and leverage high level workings
  - Admittedly, this does take some practice to get used to when reading
- Up to now, there has been a high focus on managing tuples and lists as the major data structures for holding data.
- Further on, the course will explore additional ways of storing data, leading to
  - more efficient, i.e., faster, algorithms
  - more interesting algorithms
  - more opportunities to explore and leverage abstraction
  - more fun!
- We have seen one example with the *key value store*
  - We used a list, but that it not directly revealed in the type
  - Abstract operations for creating a KVS, adding, finding and deleting keys
  - We could use another data structure without changing the abstract operation
  - The actual representation is *hidden*