More about Lists

Polymorphism vs. Overloading

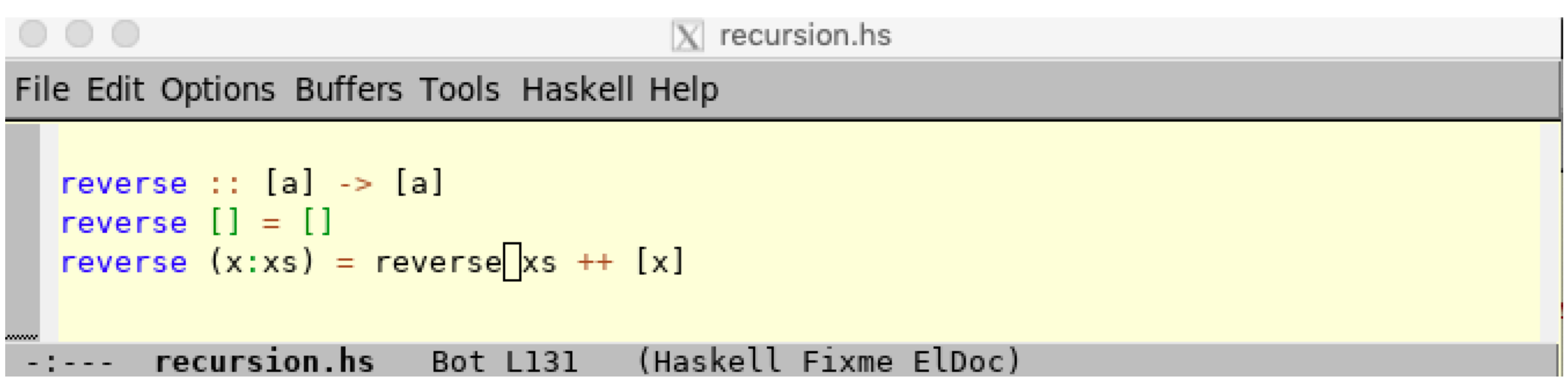
- We have seen a number of list functions where the type include a *type variable*
  - `length :: [a] -> Int`
  - `take :: Int -> [a] -> [a]`
- We have also seen functions that operate on *different types*
  - < can be used to compare
    - numbers
    - characters
    - strings (lists)
- What's the difference?
  - The first form is called *polymorphism* where we *don't care* about the type
  - The second form is called *overloading* where we, under the hood, have *different implementations* for each type supported
- The existence of both makes writing code very convenient
  - The alternative would lead to a lot of repeated, very similar code
- More details later in the course

Additional List Functions

- How do get the nth element of a list?
  - We count (index) from zero
  - [0, 1, 2, 3, 4]
- Base case: index is zero, i.e., we want the first element
  - result is head of list
- Inductive case:
  - We want element N, N >0
  - get element N-1 in tail of list
- Already defined as infix `!!`
  - `[0, 1, 2, 3] !! 2 => 2`

Additional List Functions

- We mentioned the existence of function to reverse a list
  - `reverse [1] => [1]`
  - `reverse “paris” => “sirap”`
  - `reverse [“to”, “be”, “or”] => [“or”, “be”, “to”]`
- How can we implement this?
- Base case: empty list
  - result empty list
- Inductive case
  - reverse tail of list (recursive call)
  - add head to end of reversed tail (`append` in last lecture, `++` predefined)
    - `[1,2] ++ [3] => [1,2,3]`



Reverse

recursion.hs

File Edit Options Buffers Tools Haskell Help

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

-:--- recursion.hs   Bot L131   (Haskell Fixme ElDoc)

- Simple and straight forward
- What happens during evaluation?
- For a non empty list we have
  - one recursive call to `reverse`
  - one call to `++`
- For `++` we have
  - one recursive call to `++`
  - one call to `:`
- Do the math! (later)
  - expensive

```
reverse [1, 2, 3, 4]
 (reverse [2, 3, 4]) ++ [1]
 ((reverse [3, 4]) ++ [2]) ++ [1]
 (((reverse [4]) ++ [3]) ++ [2]) ++ [1]
 ((((reverse []) ++ [4]) ++ [3]) ++ [2]) ++ [1]
 ((([] ++ [4]) ++ [3]) ++ [2]) ++ [1]
 (([4] ++ [3]) ++ [2]) ++ [1]
  ((4 : ([] ++ [3])) ++ [2]) ++ [1]
  ((4 : [3]) ++ [2]) ++ [1]
 ([4, 3] ++ [2]) ++ [1]
  (4 : ([3] ++ 2)) ++ [1]
  (4 : (3 : [] ++ [2])) ++ [1]
  (4 : (3 : [2])) ++ [1]
 [4, 3, 2] ++ [1]
  4 : ([3, 2] ++ [1])
  4 : (3 : ([2] ++ [1]))
  4 : (3 : (2 : ([] ++ [1])))
 [4, 3, 2, 1]
```

Reverse (faster)

- As before, we can use an accumulator to solve it in another way
  - Generate the reverse list as we go along
  - Solve the generalised problem of already having reversed a prefix of a list
  - L = [1, 2, 3, 4, 5, 6]
    - reverse of [1, 2, 3] is [3, 2, 1]
    - Given [4, 5, 6] and the reversed prefix [3, 2, 1]
    - New reversed prefix is 4 : [3, 2, 1] = [4, 3, 2, 1]
    - add head to reversed prefix and move along
- Base case: empty list
  - result is reversed prefix
- inductive case
  - as above - add head to reversed prefix
- What is the reversed prefix when we start?
  - the empty list
- One recursive call and one call to **:**

```haskell
areverse :: [a] -> [a]
areverse list =
  let rev [] rp = rp
      rev (x:xs) rp = rev xs (x:rp)
  in
    rev list []
```

Joining Two Lists

- Write a function that given two lists, constructs a new list of pairs, where each each part of the pair comes from either list
- `ziplists [1,2,3,4] [“one”,”two”,”three”,”four”] => [(1,”one"),(2,"two"),(3,"three"),(4,"four")]`
- Return empty list if either list is empty
  - Two base cases
- Inductive case
  - both lists are non empty
  - zip tails of lists
  - add pair of heads
- Exists as `zip`

recursion.hs

File Edit Options Buffers Tools Haskell Help

```haskell
ziplists :: [a] -> [b] -> [(a,b)]
ziplists [] _ = []
ziplists _ [] = []
ziplists (x:xs) (y:ys) =
  (x,y) : ziplists xs ys
```

-:--- recursion.hs 52% L81 (Haskell Fixme ElDoc)

Split list of pairs

- Given a zipped list (list of pairs) construct a pair of lists, i.e., essentially going in the reverse direction of `ziplists`
- `unziplist :: [(a, b)] -> ([a], [b])`
- base case: empty list
    - pair of two empty lists
- inductive case: non empty list, head is a pair
    - call recursively on tail of list, get a pair of lists
    - construct a new pair, by adding each part of the head
- exists as `unzip`

recursion.hs

File Edit Options Buffers Tools Haskell Help

```
unziplists :: [(a, b)] -> ([a], [b])
unziplists [] = ([], [])
unziplists ((x, y):pairs) =
  let (xs, ys) = unziplists pairs
  in
    (x:xs, y:ys)
```

-:--- recursion.hs 54% L90 (Haskell Fixme ElDoc)

Don't Repeat Yourself

- As should be apparent a great deal of recursive list functions follow a common pattern
  - base case is empty, often mapping to an empty list
  - inductive case is a non empty list with
    - simple recursive call on tail
    - some manipulation of head
      - apply some function, or
      - select whether to include element
- Why write the same thing over and over again?
- There exists nifty functions that
  - apply a function to each element in a list
  - keep only elements for which a condition is true
- In both cases, we have functions that takes another function as an argument

Apply function to each element

- `map :: (a -> b) -> [a] -> [b]`
- First argument is a function
- Second argument is a list

```
λ> add1 x = x+1
add1 :: Num a => a -> a

λ> map add1 [0,1,2,3,4,5,6]
[1,2,3,4,5,6,7]

λ> map length ["lambda", "calculus", "is", "all", "you", "need"]
[6,8,2,3,3,4]

λ> map isPrime [1,2,3,4,5,6,7,8,9,10,11,12,13]
[False,True,True,False,True,False,True,False,False,False,True,False,True]

λ> map odd [1,2,3,4,5,6,7,8,9,10,11,12,13]
[True,False,True,False,True,False,True,False,True,False,True,False,True]

λ> mod2 n = mod n 2
mod2 :: Integral a => a -> a

λ> map mod2 [1,2,3,4,5,6,7,8,9,10,11,12,13]
[1,0,1,0,1,0,1,0,1,0,1,0,1]
λ>
```

Filter elements with property

- `filter :: (a -> Bool) -> [a] -> [a]`
- First argument is a function return a Bool, called a predicate
- Keep the elements for which the predicate is true

```
λ> filter isPrime [1,2,3,4,5,6,7,8,9,10,11,12,13,14]
[2,3,5,7,11,13]

λ> ispos n = n > 0
ispos :: (Ord a, Num a) => a -> Bool

λ> filter ispos [-2,-3,1,0,3,-1,7,8,8]
[1,3,7,8,8]

λ> filter odd [1,2,3,4,5,6,7,8,9,10]
[1,3,5,7,9]

λ> ispalindrom l = reverse l == l
ispalindrom :: Eq a => [a] -> Bool

λ> filter ispalindrom ["abba", "haskell", "rotor", "motor"]
["abba","rotor"]

λ>
```

List Comprehension

- Haskell provides a convenient shorthand notation for application and filtering (at the same time)
- In mathematics there is notation for set comprehension
  - $\{x^2 \mid x \in N,\ x \bmod n == 0\}$
- For lists we can use some very similar
  - `[x*x | x <- [1,2,3,4,5,6,7], mod x 2 == 1]`
  - Square the odd numbers from the given list
- We can have several lists and several predicates/filters
  - we take a combination of the lists
  - all predicates must be true
- The general form is
  - $[\ f\ v_0..v_n \mid v_0 \texttt{<-}\ l_0,\ v_1 \texttt{<-}\ l_1,\ ..\ v_n \texttt{<-}\ l_n,\ p_o\ v_0..v_n,\ p_1\ v_0..v_n,\ ..]$

List Comprehension

```
λ> [x*x | x <- [1,2,3,4,5,6,7], odd x]
[1,9,25,49]

λ> [x*x | x <- [1,2,3,4,5,6,7], isPrime x]
[4,9,25,49]

λ> [x*x | x <- [1,2,3,4,5,6,7], isPrime x, odd x]
[9,25,49]

λ> [(x,y) | x <- [1,2,3], y <- ['x','y']]
[(1,'x'),(1,'y'),(2,'x'),(2,'y'),(3,'x'),(3,'y')]

λ> [(x,y) | x <- [1,2,3], y <- ['x','y'], x > 1]
[(2,'x'),(2,'y'),(3,'x'),(3,'y')]

λ> [(4*x,y) | x <- [1,2,3], y <- ['x','y','z'], x > 1, y < 'z']
[(8,'x'),(8,'y'),(12,'x'),(12,'y')]

λ>
```

List Ranges

- Nice exercises:
  - Write a function `between N M` that generates a list with integers:
    - `[N, N+1, .. M-1, M]`
    - `between 1 7 => [1,2,3,4,5,6,7]`
  - Write a function `betweenStep N L M` that generates a list with integers:
    - `S = L-N`
    - `[N, N+S, N+2*S, .. N+I*S]` (where $N+I*S \leq M$, $N+(I+1)*S>M$)
    - `betweenStep 1 3 8 => [1,3,5,7]`
- Haskell has short hands for these
  - `[1..7] => [1,2,3,4,5,6,7]`
  - `[1,3..8] => [1,3,5,7]`
- Haskell also has the short hands where you omit the upper limit, i.e.,
  - `[0..]`
  - `[1,4..]`
  - These mean what you would expect, i.e., lists of all numbers from 0 and from 1 in increments of 3.

List Ranges

- *All* numbers..?
  - How can that be?
  - Wouldn't that take a large amount of time and space to generate?
  - Yes, but only if we actually do it.
  - We can be lazy..
  - .. and bases cases are so tricky anyway..

```
from :: Integer -> [Integer]
from n = n : from (n+1)
```

Lazy Evaluation and Lists

- We have seen that, e.g., boolean operators `&&` and `||` are lazy
  - They only evaluate as much is needed to know the result
  - This is common for other languages as well
- Haskell is *fully lazy*, i.e., nothing is evaluated until it is really needed
  - This is *not* common for other languages
  - The basic list constructor : is thus not evaluated until someone actually "asks" for the tail of list
  - Note that printing something qualifies as "asking" for the value
- A call to `from 0` will just setup a "recipe" for constructing a list of all natural numbers, but produce them only when asked for
- Since `:` is the only way to construct a list, this property holds for *all* functions that constructs lists

Lazy Evaluation and Lists

```
λ> nats = from 0

λ> primes = filter isPrime nats

λ> take 10 nats
[0,1,2,3,4,5,6,7,8,9]

λ> take 10 primes
[2,3,5,7,11,13,17,19,23,29]

λ> square n = n*n

λ> squares = map square nats

λ> take 10 squares
[0,1,4,9,16,25,36,49,64,81]

λ> take 10 (filter odd squares)
[1,9,25,49,81,121,169,225,289,361]

λ> add2 x = x+2

λ> evens = 0 : map add2 evens

λ> take 10 evens
[0,2,4,6,8,10,12,14,16,18]
```