Recursion over lists
COM1022 Functional Programming Techniques

Dr Hans Georg Schaathun
University of Surrey

Spring 2010
Week 1

Learning Objectives

- Refresh key concepts covered in the Autumn
  - Lists and tuples
  - Recursion
- Learn to apply recursion to lists.

Recursion

What is recursion

- A problem of size $n$
- Base case: solution for small $n$ (e.g. $n = 0$)
- Recursive case:
  - Assume a known solution for $n - 1$
  - Provide a solution for $n$
- Nine out of ten applications of recursion are primitive recursion

Example

Powers of two

- Calculate powers of two $2^n$
- $\text{power2 } 0 = 1$
- $\text{power2 } n = 2 \times \text{power2 } (n-1)$
Other simple applications of recursion

You should be able to use recursive cases based on solutions for other smaller values of $n$.

- based on solution for $n/2$ (instead of $n - 1$)
- as we have seen in exponentiation (split and conquer)
- needing solutions more than one value of $n$
- e.g. $n - 1$ and $n - 2$, as in the Fibonacci numbers

Example

Parity of an integer

- `iseven i :: Int -> Bool`
- `iseven 0 = True`
- `iseven 1 = False`
- `iseven i = iseven (i-2)`

Advanced forms recursion

Some advanced forms of recursion are out of scope for this course.

- Extra auxiliary functions required
  - as in the fast Fibonacci sequence of Thompson (p. 76)
- Mutual recursion:
  - two functions mutually call each other, on smaller problems
Problem solving
Using recursion

- Using recursion in practice is an exercise of creativity
  - no hard and fast rules
- Solve the problem **by hand** for small values of $n$
  - see if you can work out a pattern
  - then formalise it
- Try primitive recursion
  - primitive recursion applies most often

The Towers of Hanoi

- The Towers of Hanoi is a famous puzzle
- A stack of disks in decreasing size
- A disk may never be placed on top of a smaller disk
- You are only allowed to move one disk at a time
- You have three positions which can hold a stack of disks
- **Task:** Move the tower from Position 1 to Position 2
- How many moves do you require (optimal solution)?
  - Try it at [http://mazeworks.com/hanoi/index.htm](http://mazeworks.com/hanoi/index.htm)

Lists and Tuples

Summary

- For any type $a$, there is a list type $[a]$
  - The list holds an **arbitrary** number of elements
  - All elements of the **same** type $a$
    - e.g. list of integers: $[	ext{Int}]$
- For any selection of types $a, b, \ldots z$
  - There is a tuple type $(a, b, \ldots z)$
  - **Specified number of elements**
  - May include **different** types
  - Specified types in specified order

List comprehension

- $[\text{expr } x | x <- l, \text{cond } x]$
  - $\text{expr } x$ – an arbitrary expression in $x$
  - $x <- l$ – $x$ runs through the values of the list $l$
  - $\text{cond } x$ – a boolean expression
- Calculate $\text{expr } x$
  - for all $x$ in $l$
    - such that $\text{cond } x$ is True
- Collect all the results in a list
- Just like set comprehension
  - $\{e(x)|x \in \ell, c(x)\}$
Examples
List comprehension

- All divisors of \( n \)
  - \( \text{divisors} \ n = [x | x \leftarrow [1..n], n \mod x = 0] \)
- All the names from (name, age) pairs
  - \( \text{names} :: ([\text{String}, \text{Int}]) \rightarrow [\text{String}] \)
  - \( \text{names} \ l = [x | (x, y) \leftarrow l] \)

2-D list comprehension
Another example

- \( [ (i, j) | i \leftarrow [1..3], j \leftarrow [1..3] ] \)
- Set of co-ordinates in a \( 3 \times 3 \) grid
  - \( [(1,1), (1,2), (1,3), (2,1), (2,2), (2,3), (3,1), (3,2), (3,3)] \)
- Note: major index first
  - i.e. all the \( i = 1 \) entries come first

A database example

- Consider a library catalog
- Each book has one \textit{record}
  - recording \textit{author}, \textit{title}, \textit{year}, etc.
- The catalogue is a collection of records

The record

- What data type can we use for a record?
  - Author – \textit{String}
  - Title – \textit{String}
  - Year – \textit{Int}
- The tuple lets us combine multiple fields into a record
  - \( (\text{String}, \text{String}, \text{Int}) \)
The catalogue

- What data type can we use for the catalogue?
- A list
- What list type?
- It is a collection of records – remember the type used for a record
- \[ (\text{String, String, Int}) \]

Some sample data

- A book
- A catalogue

Functions on records

- Selector function:
  - \( \text{title} (x,y,z) = y \)
- Pretty print:
  - \( \text{pretty} (x,y,z) = y ++ " by " ++ x ++ " (" ++ show z ++ ")" \)

Sample> title simon
"The Craft of Functional Programming"
Sample> pretty simon
"The Craft of Functional Programming by Simon Thompson (1999)"

Functions on the catalog

- Add record
  - \( \text{addbook} :: \ [ (\text{String, String, Int}) ] \to (\text{String, String, Int}) \to [ (\text{String, String, Int}) ] \)
    \[ \text{addbook} \ l \ x = x:l \]
- Remove record
  - \( \text{removebook} :: \ [ (\text{String, String, Int}) ] \to (\text{String, String, Int}) \to [ (\text{String, String, Int}) ] \)
    \[ \text{removebook} \ l \ x = [ y | y \not\in \ l, x != y ] \]
- Extract books by a given author
  - \( \text{byauthor} :: \ [ (\text{String, String, Int}) ] \to \text{String} \to [ (\text{String, String, Int}) ] \)
    \[ \text{byauthor} \ l \ a = [ (x,y,z) | (x,y,z) \not\in \ l, x = a ] \]
Recursion over lists

**Base case** which is our small case?
- Empty list []
- or (maybe) list of size 1 [x]

**Recursive case** which smaller cases can we use to help us?
- List of size one smaller
- or (maybe) list of half the size

**For example**
- foo\textbackslash{}bar :: \[a\] \rightarrow b
- foo\textbackslash{}bar [] = 0
- foo\textbackslash{}bar x:xs = foo x + foo\textbackslash{}bar xs

The list constructor

- The list constructor:
  - x:xs prepends x to the start of xs
- Unique decomposition of lists
  - x1:x2:x3:x4:...:xn:[]
- Note that decomposition with ++ is not unique
  - x:xs likely faster than [x] ++ xs
- Unique decomposition gives convenient pattern matching
  - foo\textbackslash{}bar [] (base case)
  - foo\textbackslash{}bar x:xs (recursive case)
    - xs is a list of length one smaller
    - x is a single element

The length of a list

**An example**

- What is the length of a list l?
  - length :: [a] \rightarrow Int
- Base case
  - length [] = 0
- Recursive case
  - length x:xs = 1 + length xs

The sum of a list of numbers

**Another example**

- Add all the integers in a list
  - sum :: [Int] \rightarrow Int
- Base case
  - sum [] = 0
- Recursive case
  - sum x:xs = x + length xs
List folding

The `foldr1` function

- Take a list: \([e_1, e_2, e_3, \ldots, e_n] :: [a]\)
  - and a function: \(f :: a -> a -> a\)
- We often need to fold \(f\) into the list:
  \[e_1 \ 'f' \ ( e_2 \ 'f' \ ( e_3 \ 'f' \ldots \ ( \ldots \ 'f' \ e_n ) \ldots ) )\]
  - e.g. `sum` is folding of `+`
- Folding is a built-in function
  - `foldr1 f :: ( a -> a -> a ) -> [a] -> a`
  - `sum = foldr1 (+)`

Unsurprisingly, defined by recursion

- `foldr1 f [x] = x`
- `foldr1 f (x:xs) = f x (foldr f xs)`
- Note `foldr1 f []` is undefined

Left and right folding

`foldl1` and `foldr1`

- The 'r' in `foldr1` is for right
  - Folding is right to left
  - The function \(f\) is applied first to the two right-most elements
  - ... then to the result and the third element from the right, etc.
- There is a left folding function too: `foldl1`
- Clearly, this matters only for non-associative operators
  - `foldr1 (+)` is the same as `foldl1 (+)`

More general folding

- `foldr :: ( a -> a -> a ) -> a -> [a] -> a`
- Input arguments
  - Combining function
  - Starting value for the folding
  - List of values to combine
- `foldr f s [] = s`
- `foldr f s (x:xs) = f x (foldr f s xs)`
- `foldr (+) s [e_1, e_2, \ldots, e_n] = e_1 + e_2 + \ldots + e_n + s`
- Even more generally, different base types
  - `foldr :: ( a -> b -> b ) -> b -> [a] -> b`
Applications of folding

- Many standard mathematical operations can reasonably be folded
  - Maximum, minimum
  - Sum (+)
  - Product (*)
  - String/List concatenation (++)

Three ways to reverse a list

- Consider a list \( l \)
- Task: reverse the order of the elements
- Typical application of recursion.
- How do we split the list for recursion?
- Three options
  1. split off the last element
     - \( \text{last } l - \text{init } l \)
  2. split off the last element
     - \( \text{head } l - \text{tail } l \)
  3. split the list in half
     - \( \text{take } b l - \text{drop } b l \)
     - where \( b \) is half the length of \( l \)

The base case

- What do you propose as a base case?
- \( \text{rev } [ ] = [ ] \)
- \( \text{rev } [x] = [x] \)
Examples

Recursion based on a list of length $n/2$

- Split the list in two equal halves
  - Use `drop (n\div2) l` and `take (n\div2) l`
- Reverse each half (length $n/2$)
- Swap and concatenate

- Advantages of this approach?
  - Fewer steps before you reach the base case
  - Halving reduces the size faster than removing one

Summary

- Exercises are based on recursion
  - The Towers of Hanoi
  - Recursion on integers
  - Simple recursion on lists
- Recursion is a fundamental concept with many applications
  - Practice,
    - Learn to see the variety of applications of recursion
- and, of course, get used to the syntax of lists in Haskell