COMP0020 Functional Programming

Lecture 0

Designing Functional Programs

Contents

- Answer to exercise
- Currying and partial applications
- Approaches to design
- Case analysis : example "reverse"
- Structural induction : example "startswith"

Stack recursion answer


```
|| threes is a function that takes a list of
|| numbers and returns the number of
|| occurrences of the number 3 in that list
threes :: [num]->num
threes []           = 0
threes (3 : rest)   = 1 + (threes rest)
threes (x : rest)   = threes rest
```

Accumulative recursion answer


```
|| Alternative definition :
fours :: [num] -> num
fours input = xfours (input, 0)
              where
              xfours ([], a)          = a
              xfours (4 : rest, a)    = xfours (rest, a + 1)
              xfours ((x : rest), a)  = xfours (rest, a)
```

Currying

- Functions that take more than one argument
  - Either collect arguments into a tuple
  - Or use "curried" definition (named after Haskell B. Curry)
- A curried definition of a function that takes two arguments does **not** use a tuple. Compare curried function f and uncurried function g :

$$f\ x\ y\ =\ x\ +\ y$$

$$g\ (x, y)\ =\ x\ +\ y$$

  - compare f with the lambda expression $\lambda x.(\lambda y.(x + y))$
- The types of these functions are also different :

$$f :: num -> num -> num$$

$$g :: (num, num) -> num$$

- Application is also different : $(f\ 3\ 4)$ compared with $g\ (3, 4)$

Curried accumulative version


```
|| Alternative definition
fives :: [num] -> num
fives input = xfives input 0
              where
              xfives []       a = a
              xfives (x : rest) a = xfives rest (a + 1)
              xfives (x : rest) a = xfives rest a
```

Partial Applications

- Only for curried functions

$$f :: num -> num -> num$$
$$f\ x\ y = x + y$$

- We can *partially apply* the function f to its first argument, and the result is a function of one argument :

$$Miranda\ (f\ 3) :: num -> num$$

- We can give a name to a partial application :

$$fred = (f\ 3)$$
$$main = fred\ 4$$

Partial Applications of operators

- Operators have curried definitions and can also be partially applied:

$$\begin{array}{l} \textit{Miranda} \quad (+) \ :: \\ num -> num -> num \\ \\ \textit{Miranda} \quad (+9) \ :: \\ num -> num \end{array}$$

- Operator sections :
  - Operator pre-sections : $(3\ +)\ x \equiv (3\ +\ x)$, $(2\ *)\ x \equiv (2\ *\ x)$, $(5\ -)\ x \equiv (5\ -\ x)$, $(9\ /)\ x \equiv (9\ /\ x)$
  - Operator post-sections : $(+\ 3)\ x \equiv (x\ +\ 3)$, $(*\ 2)\ x \equiv (x\ *\ 2)$, $(/\ 9)\ x \equiv (x\ /\ 9)$

- There is no post-section for the subtraction operator, because (-3) applies the unary minus operator to 3 to give negative 3.

Approaches to design

- Case Analysis (see the book, Section 3.7.1)
  - consider what VALUES can occur as input to a function
  - use PATTERN MATCHING to match exact values or conditionals for relational tests

- Structural Induction (see Section 3.7.2)
  - Helps you to write the "looping" part of a recursive function

Case Analysis

- consider the function "myreverse", which takes a list of anything and returns the list with all elements reversed

$$
\begin{aligned}
&myreverse :: [*] -> [*] \\
&myreverse\ [] &&= [] \\
&myreverse\ (x : []) &&= (x : []) \\
&myreverse\ (x : (y : [])) &&= (y : (x : [])) \\
&myreverse\ (x : (y : (z : []))) &&= (z : (y : (x : []))) \\
&myreverse\ (x : rest) &&= ????
\end{aligned}
$$

Case Analysis : "reverse"

- To design the looping part of the function requires some thinking
- Look at the cases and their solutions and try to find a common theme
- In this case "reverse the rest of the list and put x at the end" :

$$myreverse\ (x : rest)\ =\ (myreverse\ rest)\ ++\ [x]$$

- NB in the above code it is essential to put the item x inside a list (to create $[x]$) because the function $++$ takes two lists as arguments. Compare this with the operator : ("cons") which takes an element and a list of elements.

Case Analysis : "reverse"

- Final version :

$$
\begin{array}{lll}
myreverse :: [*] & -> & [*] \\
myreverse\ [] & = & [] \\
myreverse\ (x : rest) & = & (myreverse\ rest) + + [x]
\end{array}
$$

Structural Induction

- Induction versus Deduction
- Induction versus Structural Induction
  - Induction on Lists
- Base Case
- Induction hypothesis
- Inductive step - you still need to do some thinking!

Induction on Lists : "startswith"

- The function "startswith" takes a two-tuple containing two lists of anything and returns True if the second list starts with the first list (otherwise it returns False)
- E.g.
  - startswith ([1,2], [1,2,3,4]) returns True
  - startswith ([1,2], [2,3,4]) returns False

Induction on Lists : "startswith"

- Design Steps :
- Specify the TYPE
- Consider the General Case (the part that loops) before considering the base case
  - This helps to identify the parameter of recursion !
- Consider the base case(s)

Induction on Lists : "startswith"

- Type

$$startswith :: ([*], [*]) -> bool$$

- General case

$$startswith\ ((x : xs), (y : ys)) = ???$$

  - Possible induction hypotheses :

$$startswith\ (xs, (y : ys))$$
$$OR : startswith\ ((x : xs), ys)$$
$$OR : startswith\ (xs, ys)$$

  - Which one of the above helps us to define $startswith\ ((x : xs), (y : ys))$?

Induction on Lists : "startswith"

- General case

$$startswith\ ((x : xs),\ (y : ys)) = ??$$

Use induction hypothesis : $startswith\ (xs,\ ys)$

$$startswith\ ((x : xs),\ (y : ys))$$
$$= True,\ if\ (x = y)\ \&\ startswith\ (xs,\ ys)$$
$$= False,\ otherwise$$

Induction on Lists : "startswith"

- General case

Or, simply :

$$startswith\ ((x : xs),\ (y : ys)) = (x = y) \,\&\, startswith\ (xs,\ ys)$$

Induction on Lists : "startswith"

- Base cases(s)
  Because there are two parameters of recursion, we must consider two base cases :
  startswith ([], any)
  and
  startswith (any, [])

Induction on Lists : "startswith"

- Base case(s)
- For the first base case, there is no obvious "right" or "wrong" solution – we choose to return True

$$startswith\ ([],\ any)\ =\ True$$

- For the second base case the result is obvious :

$$startswith\ (any,\ [])\ =\ False$$

Induction on Lists : "startswith"

- The final solution is :


```
startswith :: ([*], [*]) -> bool
startswith ([], any)          = True
startswith (any, [])          = False
startswith (x:xs, y:ys)       = (x == y) & startswith (xs, ys)
```

Summary

- Answer to exercise
- Approaches to design
- Case analysis : example "reverse"
- Structural induction :
  - Induction hypothesis
  - Inductive step
  - Base case(s)
- Example "startswith"

END OF LECTURE