COMP0020 Functional Programming

Lecture 8

Designing Functional Programs

(2)

Contents

- Structural induction : example “append”
- Passing data between functions : “isort”
- Modes of recursion : tail recursion and mutual recursion
- Removing mutual recursion
- Lazy evaluation : infinite lists

Induction on Lists : "append"

- The "append" function takes two lists of anything and returns a single list consisting of all the elements of the first list, followed by all the element of the second list
- Type :
  append : : ([*], [*]) -> [*]
- Possible Induction hypotheses :

$$append\ (xs,\ (y : ys))$$

$$OR : append\ ((x : xs),\ ys)$$

$$OR : append\ (xs, ys)$$

to help define the general case :

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

Induction on Lists : "append"

- Think about what each possible induction hypothesis would give you
- For example, if we want to define the general case for append ([1,2,3], [4,5,6]) :

$append(xs, (y : ys))$	*gives*	$[2, 3, 4, 5, 6]$ – *does that help*?
$append((x : xs), ys)$	*gives*	$[1, 2, 3, 5, 6]$ – *does that help*?
$append(xs, ys)$	*gives*	$[2, 3, 5, 6]$ – *does that help*?

Induction on Lists : "append"

- Answer : use append (xs, (y :ys)) as the induction hypothesis. Thus, there is only one parameter of recursion. The general case is :

$$\text{append } ((x :xs), (y :ys)) = x : (\text{append } (xs, (y :ys)))$$

- Or, simply :

$$\text{append } ((x :xs), any) = x : (\text{append } (xs, any))$$

Induction on Lists : “append”

- Base case (for parameter of recursion) :

$$\text{append } ([], (y : ys)) = ???$$

- We choose the answer to be (y :ys)

$$\text{append } ([], (y : ys)) = (y : ys)$$

- Or, simply :

$$\text{append } ([], any) = any$$

Induction on Lists : “append”

- Final solution :

$$
\begin{aligned}
&append :: ([*], [*]) \rightarrow [*] \\
&append\ ([],\ any) = any \\
&append\ ((x : xs),\ any) = x : (append\ (xs,\ any))
\end{aligned}
$$

Passing data between functions

- A functional program usually contains several (many) functions
- Focus on how data passes between those functions
- Example : insertion sort “isort”

Insertion Sort (specification)

- Define "sorted list"
  - An empty list is already sorted
  - A singleton list is already sorted
  - The list (x :xs) is sorted if
    - x is less than all items in xs, AND
    - xs is sorted
- NB only lists of numbers

Insertion Sort (strategy)

- Start with two lists A and B
- A is the input list
- B is initially empty
- One at a time, move an element from A to B
- Ensure that at all times B is sorted
  - We will need a function that can insert a number into a sorted list and return a sorted list

Insertion Sort (design)

- The list B is an accumulator
  - So use accumulative recursion
- Top-down approach : assume the function "insert" exists and design the rest of the program first (leap of faith !)
- Then design "insert"

Insertion sort


```
||comments ...
isort :: [num] -> [num]
isort any = xsort any []

||comments ...
xsort :: [num] -> [num] -> [num]
xsort [] sorted = sorted
xsort (x : xs) sorted = xsort xs (insert x sorted)
```

Insertion sort

- Code for "insert"


```
||comments...
insert :: num -> [num] -> [num]
insert x []          = [x]
insert x (y : ys)    = (x : (y : ys)), if (x < y)
                     = ???
```

Insertion sort (code 3)

- Use induction hypothesis : assume that "insert x ys" correctly inserts x into the list ys and produces the correct sorted list as a result :

$$
\begin{aligned}
&\textit{insert} :: \textit{num} \to [\textit{num}] \to [\textit{num}] \\
&\textit{insert}\ x\ [] &&= [x] \\
&\textit{insert}\ x\ (y : ys) &&= (x : (y : ys)),\ \textit{if}\ (x < y) \\
&&&= y : (\textit{insert}\ x\ ys),\ \textit{otherwise}
\end{aligned}
$$

Insertion sort — full code


```
isort :: [num] -> [num]
isort any = xsort any []

xsort :: [num] -> [num] -> [num]
xsort []       sorted = sorted
xsort (x : xs) sorted = xsort xs (insert x sorted)

insert :: num -> [num] -> [num]
insert x []       = [x]
insert x (y : ys) = (x : (y : ys)), if (x < y)
                  = (y : (insert x ys)), otherwise
```

More modes of recursion

1 : tail recursion

$$
\begin{aligned}
&mylast :: [*] \rightarrow * \\
&mylast\ [] \quad = \ error\ \text{“}no\ last\ item\ of\ empty\ list\text{”} \\
&mylast\ (x : []) \ = \ x \\
&mylast\ (x : xs) = mylast\ xs
\end{aligned}
$$

More modes of recursion

2 : mutual recursion

$$
\begin{aligned}
&nasty :: [char] \quad -> [char] \\
&nasty\ [] \quad = [] \\
&nasty\ (')' : rest) \quad = xnasty\ rest \\
&nasty\ (x : xs) \quad = (x : (nasty\ xs))
\end{aligned}
$$

$$
\begin{aligned}
&xnasty :: [char] \quad -> [char] \\
&xnasty\ [] \quad = error\ \text{“missing end bracket”} \\
&xnasty\ (')' : rest) = nasty\ rest \\
&xnasty\ (x : xs) \quad = xnasty\ xs
\end{aligned}
$$

Removing mutual recursion


```
skip :: [char] -> [char]
skip []       = []
skip ('(':rest) = skip (doskip rest)
skip (x:rest)   = x : (skip rest)

doskip :: [char] -> [char]
doskip []       = error "missing end bracket"
doskip (')':rest) = rest
doskip (x:xs)     = doskip xs
```

Lazy evaluation : infinite lists

- Lazy evaluation of function arguments
  - Evaluate fst (24, (37 / 0))
  - Remember definition of fst :
    ```
    fst : : (*,**) -> *
    fst (x,y) = x
    ```
- Lazy evaluation of data constructors (e.g. the "cons" operator for lists)
  - Some forms of "bad" recursion may NOT result in infinite execution because lazy evaluation of data constructors means that they are evaluated ONLY AS FAR AS NECESSARY :
    ```
    f : : num -> [num]
    f x = (x : (f (x + 1)))
    main = hd (tl (f 34))
    ```
  - Another example :
    ```
    ones = (1 : ones)
    main = hd (tl (tl ones))
    ```

Summary

- Structural induction : example "append"
- Passing data between functions : example "isort"
- Modes of recursion : tail recursion and mutual recursion
- Removing mutual recursion
- Lazy evaluation : infinite lists

END OF LECTURE