Functional Programming

Christopher D. Clack

FUNCTIONAL PROGRAMMING

Lecture 9

DESIGNING FUNCTIONAL PROGRAMS

FUNCTIONAL PROGRAMMING
DESIGNING FUNCTIONAL PROGRAMS

This lecture continues the exploration of approaches to designing functional programs. It first returns to the "structural induction" approach that was explained in Lecture 6 , and provides another practical example. The passing of data between functions is explored through the practical example of a sorting algorithm. Recursion is revisited, and in particular there will be a focus on mutual recursion and why it can be problematic, followed by an example of how to remove mutual recursion. The lecture ends with a discussion of lazy evaluation and computation over infinite lists

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

FUNCTIONAL PROGRAMMING
DESIGNING FUNCTIONAL PROGRAMS

INDUCTION ON LISTS – "append" example

1. Type:

   append :: [*] -> [*] -> [*]

2. Induction Hypotheses and implied parameter(s) of recursion (POR):

   append xs (y:ys) || POR is xs

   append (x:xs) ys || POR is ys

   append xs ys || POR is xs and ys

   To be used inside the function body for the most general equation defining the function:

   append (x:xs) (y:ys) = ??????

Induction on lists: "append" (see Page 88 of the Miranda book)

The "append" function mimics the functionality of the built-in operator "++", which 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

First we state the type of our "append" function. We will use a Curried definition:

append :: [*] -> [*] -> [*]

Next, we explore the possible Induction Hypotheses and their implied parameter(s) of recursion (POR). The recursive call could be any one of these three alternatives:

append xs (y:ys) || POR is xs

append (x:xs) ys || POR is ys

append xs ys || POR is xs and ys

The chosen recursive call will be used inside the function body for the most general equation defining the function:

append (x:xs) (y:ys) = ??????

FUNCTIONAL PROGRAMMING
DESIGNING FUNCTIONAL PROGRAMS

Example: to define the general case for
append [1,2,3] [4,5,6]
the possible recursive calls are:

append xs (y:ys) giving [2,3,4,5,6]

append (x:xs) ys giving [1,2,3,5,6]

append xs ys giving [2,3,5,6]

Which of the above is most helpful?

Answer: append xs (y:ys) which can be used like this:

append (x:xs) (y:ys) = x: (append xs (y:ys))

Or, more simply:

append (x:xs) any = x: (append xs any)

The Induction Hypothesis is that the recursive call will give the correct answer when used inside the function body. Think about what each possible recursive call would give (it helps to consider a concrete example)

For example, to define the general case for *append [1,2,3] [4,5,6] the possible recursive calls are:*

append xs (y:ys) giving [2,3,4,5,6]

append (x:xs) ys giving [1,2,3,5,6]

append xs ys giving [2,3,5,6]

Which of the above results would help most in defining the function body for

append (x:xs) (y:ys) = ??????

The answer is:

Use *append xs (y:ys)* as the recursive call. Thus, there is only one POR (the first list) and the function body for the looping part of the function would be:

append (x:xs) (y:ys)
= x:(append xs (y:ys))

Note that because y and ys never occur on their own in the function body (only as part of the expression (y:ys)) we don't need separate names for the head and tail of the second argument list, so we can give that list a single name:

append (x:xs) any = x : (append xs any)

FUNCTIONAL PROGRAMMING
DESIGNING FUNCTIONAL PROGRAMS

The base case for append

append [] (y:ys) = ??????

Let [] be the identity for append:

append [] (y:ys) = (y:ys)

Or more simply:

append [] any = any

Final solution:

```
append:: [*] -> [*]  -> [*]
append   []       any = any
append   (x:xs) any = x : (append xs any)
```

Base case for "append"

We need to consider the base case for the Parameter of Recursion, which is the first list. The base case for a list is often (though not always) the empty list, and that makes sense for this function

So what should *append [] (y:ys)* return? We choose to say that the empty list [] is the identity value for the function append – that is, it acts like 0 for + (or like 1 for *) in that append [] y =y and append x [] = x

So we have:

append [] (y:ys) = (y:ys)

Or, more simply:

append [] any = any

The final solution is therefore:

```
append:: [*] -> [*] -> [*]
append []       any = any
append (x:xs) any = x : (append xs any)
```

Notice that this solution, though neat, has poor performance for the special case *append any []* which should return *any*. If required, we can add that case explicitly (though it forces some evaluation of the second list):

```
append:: [*] -> [*] -> [*]
append []       any = any
append any    []    = any
append (x:xs) any = x : (append xs any)
```

Of course, we don't need a separate base case for *append [] []*

FUNCTIONAL PROGRAMMING
DESIGNING FUNCTIONAL PROGRAMS

Passing data between functions

Example – sorting a list of numbers

Definition of “sorted”:

- An empty list is sorted
- A singleton list is sorted
- The list (x:xs) is sorted if
  - x < all items in xs AND
  - xs is sorted

A functional program usually contains several (perhaps many) functions, and now we focus on how data passes between those functions. We use an example of an algorithm to sort a list of numbers (see Page 90 of the Miranda book):

We start with the definition of what we mean by “sorted”:

- An empty list is sorted
- A singleton list is sorted
- The list (x:xs) is sorted if
  - x < all items in xs AND
  - xs is sorted

FUNCTIONAL PROGRAMMING
DESIGNING FUNCTIONAL PROGRAMS

Strategy – Insertion Sort

- Start with two lists A and B
- A is the input list to be sorted
- B is initially empty
- One at a time, move an element from A into B
- Ensure at all times that B is sorted

Use accumulative recursion for the top-level function (call it "isort"). This will then call a lower-level function "insert"

There are many different strategies for sorting (Bubble Sort, Quick Sort etc.) – we will choose to implement Insertion Sort, which briefly works as follows:

- Start with two lists A and B
- A is the input list to be sorted
- B is initially empty
- One at a time, move an element from A into B
- Ensure at all times that B is sorted

In the above description, B is an accumulating parameter and it is natural to use an accumulative recursive approach to the function design

We will also need a function that can insert a number into the correct position in a sorted list of numbers and return the result. We will call this function "insert"

We will however approach this "top down" – we first we will design the top-level function (we'll call it "isort") assuming that the function "insert" already exists (and works correctly), and then we will design the function "insert"

FUNCTIONAL PROGRAMMING
DESIGNING FUNCTIONAL PROGRAMS

```
||comments . . .
||
isort :: [num] –> [num]
isort any = xsort any []
            where
            xsort []       sorted = sorted
            xsort (x : xs) sorted = xsort xs (insert x sorted)
```

Here is the code for the function “isort”, which takes a list of numbers and returns a sorted list of the same numbers

It has type [num] -> [num] and returns whatever value is returned by the application of the local function “xsort” to two arguments: the input list “any” and the empty list

The function xsort is a Curried function of two arguments. The first argument is the input list of numbers, potentially unsorted (this is list “A” from the strategy mentioned in the previous slide): the second argument is an accumulating parameter that is always sorted (list “B” in the previous slide)

If the first parameter is empty, xsort returns the accumulating parameter called “sorted”. The accumulating parameter is initially [], and if the input list starts off empty then the result returned is []. Pattern matching on the value [] for the first parameter also acts as the base case for recursion

If the first parameter is not empty, xsort binds the name “x” to its head and the name “xs” to its tail. It then returns whatever is returned by a recursive call to itself with the input list being one item smaller (it no longer has “x”) and with the second parameter being the result of inserting “x” into the correct position in the “sorted” list – this requires the existence of another function called “insert”

FUNCTIONAL PROGRAMMING
DESIGNING FUNCTIONAL PROGRAMS

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

Here is the code for the function "insert", which takes a number and a sorted list of numbers and returns the result of inserting the number in the correct position into the sorted list of numbers (producing a sorted list)

Inserting a number into the empty list has an obvious result (a singleton list). This list pattern also acts as the base case for recursion

In our previous definition of a sorted list we stated that the numbers would be in ascending numeric order. Thus, if "x" is less than "y" (the first item in the sorted list) then the result must simply be (x : (y : ys))

However, if "x" is bigger than the first item "y" it is not clear where to place "x" inside the sorted list "ys" (it might be at the start, and the end, or in the middle)

We can use the induction hypothesis that (insert x ys) returns a list with x embedded in ys in the correct position. If the induction hypothesis holds then the inductive step is simply to cons "y" on to the front of the result of the recursive call – thus (y : (insert x ys))

FUNCTIONAL PROGRAMMING
DESIGNING FUNCTIONAL PROGRAMS

The full version for the sorting program is given here

Full version of the code

```
||comments . . .
||
isort :: [num] –> [num]
isort any = xsort any []
            where
            xsort []       sorted = sorted
            xsort (x : xs) sorted = xsort xs (insert x sorted)

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

FUNCTIONAL PROGRAMMING
DESIGNING FUNCTIONAL PROGRAMS

MORE MODES OF RECURSION

- Tail recursion
- Mutual recursion
- Removing mutual recursion

Here we return to the topic of different modes of recursion (recalling our previous discussion of stack and accumulative recursion)

In the next slide we will briefly introduce tail recursion with an example, and say why it is important

We will then introduce an example of mutual recursion, say why (except in a few cases) it is undesirable, and show how our example of mutual recursion can be modified to remove the mutual recursion

FUNCTIONAL PROGRAMMING
DESIGNING FUNCTIONAL PROGRAMS

Tail recursion example

mylast :: [∗] –> ∗

mylast [] = *error "no last item of empty list''*

mylast (*x* : []) = *x*

mylast (*x* : *xs*) = *mylast xs*

The built-in function "last" returns the last (rightmost) element in a list. Here we define our own version of this function, which we call "mylast"

It is an error to apply this function to the empty list (which has no "last" item). When applied to a singleton list, the result is the only item in the list. This is also the base case for recursion. The general looping equation for the function uses the fact that the last item of a list with more than one item is also the last item of the tail of that list

Notice that this is neither stack recursion nor accumulative recursion

Conventional compiler technology passes arguments to a function on the run-time stack, and the function can then also use the run-time stack for its local variables. A recursive function places the new arguments on the run-time stack before making the recursive call. In this way a new "stack frame" is pushed onto the stack for each recursive call and popped from the stack as each recursive call returns its value. A Tail Recursive function makes a recursive call as its last action, returning whatever is returned from the recursive call. It therefore has no more need for its local variables, nor the parameters that were passed to it, so its own stack frame can be reused for the recursive call. This is substantially more efficient in memory (and time) and is an important compiler optimisation

FUNCTIONAL PROGRAMMING
DESIGNING FUNCTIONAL PROGRAMS

Mutual recursion example

```
nasty :: [char] -> [char]
nasty []          = []
nasty ('(': rest)  = xnasty rest
nasty (x : xs)     = (x : (nasty xs))

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

Time's up!

Here is an example of mutual recursion (see Page 96 of the Miranda book). There are two functions – "nasty" and "xnasty". They are mutually recursive because "nasty" makes a call to "xnasty" and "xnasty" includes a call to "nasty". The two functions are each recursive in their own right (they make calls to themselves) AND they additionally create a recursive looping structure that binds the two functions together. Mutual recursion may involve only two functions, or very many functions

The problems with mutual recursion are:

It is difficult to identify which of the functions is logically more important (unless indicated by the function name)

It is difficult to understand what the code does – the syntax is simple, but the semantics are not simple

It is impossible to run an effective unit test on a single function – it is necessary to test the entire mutually recursive group of functions as a single unit

There are of course exceptions, and sometimes mutual recursion provides a natural fit to the structure of the problem being modelled (one such example is writing the code for a finite state machine), however in general mutual recursion is considered undesirable

Take 2 minutes now to look at this code and try to understand what it does – if you're watching this interactively, enter your suggestion into the "chat".

FUNCTIONAL PROGRAMMING
DESIGNING FUNCTIONAL PROGRAMS

Mutual recursion example – what it does

```
nasty :: [char] -> [char]
nasty []          = []
nasty ('(': rest)  = xnasty rest
nasty (x : xs)     = (x : (nasty xs))

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

These two functions work together to process text – the function "nasty" is the dominant function. When "nasty" is applied to a list of characters, it returns a list of characters after deleting all those characters found inside parentheses (round brackets) – it also deletes the parentheses

nasty "hello again (I think) how are you"

→ "hello again how are you"

FUNCTIONAL PROGRAMMING
DESIGNING FUNCTIONAL PROGRAMS

Removing mutual recursion

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

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

The previous example of mutual recursion can be modified so that it no longer creates a looping structure and yet achieves the same result

Here the functions have been renamed "skip" and "doskip". In the second equation defining "skip" it makes a recursive call to itself, passing an argument that is the result of making a call to "doskip". The function "doskip" does not make any call to "skip", and therefore there is no mutual recursion

FUNCTIONAL PROGRAMMING
DESIGNING FUNCTIONAL PROGRAMS

Lazy evaluation and infinite lists

1. Lazy evaluation of function arguments

   Example: fst (24, (37 / 0)) does not cause an error

2. Lazy evaluation of data constructors (e.g. the (:) operator for lists) – data constructors are evaluated only as far as they are needed

```
f x = (x : (f (x + 1))
main = hd (tl (f 34))
```

Another example:

```
ones = (1 : ones)
main = hd (tl (tl ones))
```

We end this lecture with a few observations on lazy evaluation and infinite lists

1. Recall that lazy evaluation means that the arguments to a function are not evaluated if they are not needed by the function. Thus, for example,

   fst (24, (37 / 0))

   does not cause an error (it returns 24) because "fst" does not need the value of the second item in the 2-tuple

2. Lazy evaluation of data constructors (e.g. the (:) operator for lists) – data constructors are evaluated only as far as they are needed. Thus, for example, in the following code

```
f x = x : (f (x+1))
main = hd (tl (f 34))
```

   "main" returns 35 and does not cause an error even though "f 34" is an infinite list (of all integers starting at 34)

   Here's another example, where "ones" is the infinite list all of whose items are the number "1" and yet "main" does not cause an error:

```
ones = (1 : ones)
main = hd (tl (tl ones))
```

FUNCTIONAL PROGRAMMING
DESIGNING FUNCTIONAL PROGRAMS

This lecture has continued the exploration of approaches to designing functional programs. A further example of "structural induction" was given, followed by an example of how data is passed between functions. Recursion was revisited, and in particular there was a focus on mutual recursion and why it can be problematic, followed by an example of how to remove mutual recursion. The lecture ended with a discussion of lazy evaluation and computation over infinite lists

SUMMARY

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

FUNCTIONAL PROGRAMMING
DESIGNING FUNCTIONAL PROGRAMS