Functional Programming

Christopher D. Clack

FUNCTIONAL PROGRAMMING

Lecture 6

DESIGNING SIMPLE FUNCTIONS

FUNCTIONAL PROGRAMMING
DESIGNING SIMPLE PROGRAMS

This lecture explores approaches to designing simple functional programs. It starts with a discussion of various solutions to the programming exercise set at the end of the last lecture, including stack recursive and applicative recursive styles. It also covers Curried function definitions and partial applications

CONTENTS

FUNCTIONAL PROGRAMMING
DESIGNING SIMPLE PROGRAMS

STACK RECURSION ANSWER

Exercise: write a function that takes a list of numbers and returns the number of occurrences of the number 3 in that list

Stack recursive answer:

threes :: [*num*] –> *num*

threes [] = 0
threes (3 : *rest*) = 1 + (*threes rest*)
threes (*x* : *rest*) = *threes rest*

Stack recursion answer

Hopefully you will all have attempted to find a solution to the exercise set at the end of the last lecture. Some will have found it easy, others less so. Several solutions will now be discussed, using the exercise as a vehicle for discussing elements of function design

Here is an answer using the style of stack recursion, using pattern matching where the function "threes" is defined using three alternative equations

The first equation is threes [] = 0. This forces Miranda to evaluate the input just enough to see if it is (i) the empty list ("nil"), or (ii) not empty. If it is not empty, Miranda skips to check the next equation. If it is empty, the function "threes" returns 0 and does no further work. This also serves as the terminating condition for recursion

In the second equation the list pattern (3:rest) forces Miranda to evaluate the list argument just enough to check whether the first item is the value 3: if not, Miranda skips to check the next equation. If it is 3, then Miranda binds the name "rest" to the tail of the list argument and returns (1 + (threes rest)) which forces a recursive call to "threes" (this time with a smaller argument)

The third equation binds the name "x" to the first item in the list argument, binds the name "rest" to the tail of that list, and returns whatever is returned by the recursive call

FUNCTIONAL PROGRAMMING
DESIGNING SIMPLE PROGRAMS

threes :: [*num*] –> *num*

threes [] = 0
threes (3 : *rest*) = 1 + (*threes rest*)
threes (*x* : *rest*) = *threes rest*

threes [1,3,4,2,3,5]
-> *threes [3,4,2,3,5]*
-> *1 + (threes [4,2,3,5])*
-> *1 + (threes [2,3,5])*
-> *1 + (threes [3,5])*
-> *1 + (1 + (threes [5]))*
-> *1 + (1 + (threes []))*
-> *1 + (1 + (0))*
-> *1 + (1)*
-> *2*

Here we illustrate the evaluation of the stack recursive solution, when the function *threes* is applied to the list *[1,3,4,2,3,5]*

At the top we repeat the definition of threes. Then we reduce the expression threes [1,3,4,2,3,5] one step at a time

threes [1,3,4,2,3,5] matches the 3rd pattern, so we reduce to *threes rest* which is *threes [3,4,2,3,5]*

threes [3,4,2,3,5] matches the 2nd pattern, so we reduce to 1+ (*threes rest*) which is 1+(*threes [4,2,3,5])*

The + operator can't give a result without evaluating both of its arguments, so it requires *threes [4,2,3,5]* to be evaluated, which matches the 3rd pattern and reduces to *threes [2,3,5],* which is not yet fully evaluated and needs to be reduced further:

- First, to *threes [3,5]*
- Which then reduces to *1 + (threes [5])*

The second + operator also needs both arguments to be evaluated, so it forces *threes [5]* to be evaluated: firstly to *threes []* and then to *0*

So far only β reduction has occurred, but now δ reduction of the two + operators occurs. The δ rule for + is "STRICT" in its two arguments – they must both be evaluated before the δ rule can give a result

FUNCTIONAL PROGRAMMING DESIGNING SIMPLE PROGRAMS

ACCUMULATIVE RECURSION ANSWER

```
|| For variety, we choose to count the number of
|| occurrences of the number 4 instead

fours :: [num] –> num
fours x = xfours (x, 0)
          where
          xfours ([], a)          = a
          xfours ((4 : rest), a) = xfours (rest, (a + 1))
          xfours ((y : rest), a) = xfours (rest, a)
```

Accumulative recursion answer

We choose to count the number of occurrences of the number 4 instead

This solution uses accumulative recursion. The function “fours” returns whatever value is returned by the application “xfours (x,0)” and it is “xfours”, defined locally in the where block, that does the looping. Type definitions are not permitted inside where blocks

The xfours function uses pattern-matching with a list pattern nested inside a tuple pattern. It has three alternative equations, the first of which forces Miranda to evaluate the first item of the 2-tuple just far enough to check whether it is “nil”. If it is nil, the value “a” is returned (“a” is an accumulating parameter), but if it is not nil Miranda checks the second equation

The second equation evaluates the first item of the 2-tuple just enough to check whether its head is the value 4. If it is 4, Miranda binds the name “rest” to the tail of the first item and the name ”a” to the second item of the 2-tuple, and then returns whatever is returned by the recursive call of xfours (rest, (a+1)). If it is not 4, Miranda checks the third equation

The third equation binds ”y” to the head of the first item, “rest” to the tail of the first item, and “a” to the second item of the 2-tuple. It then returns whatever is returned by xfours (rest, a). Note that xfours (like threes) is applied to a smaller list each time it loops, stopping at []

FUNCTIONAL PROGRAMMING
DESIGNING SIMPLE PROGRAMS

```
fours :: [num] –> num
fours x = xfours (x, 0)
          where
          xfours ([], a)          = a
          xfours ((4 : rest), a) = xfours (rest, (a + 1))
          xfours ((y : rest), a) = xfours (rest, a)
```

```
   fours [1,3,4,2,4,5]
-> xfours ([1,3,4,2,4,5], 0)
-> xfours ([3,4,2,4,5], 0)
-> xfours ([4,2,4,5], 0)
-> xfours ([2,4,5], (0 + 1))
-> xfours ([4,5], (0 + 1))
-> xfours ([5], ((0 + 1) + 1))
-> xfours ([], ((0 + 1) + 1))
-> ((0 + 1) + 1)
-> (1 + 1)
-> 2
```

Here we illustrate the evaluation of the accumulative recursive solution, when the function *fours* is applied to the list *[1,3,4,2,4,5]*

The first step is that *fours [1,3,4,2,4,5]* reduces to *xfours ([1,3,4,2,4,5], 0)*

([1,3,4,2,4,5],0) matches the 3rd equation that defines xfours, and so the next reduction step is to *xfours (rest, a)* which is *xfours ([3,4,2,4,5], 0)*

The 3rd equation is matched again, resulting in *xfours ([4,2,4,5], 0)* whose argument matches the 2nd equation and thus reduces to *xfours (rest, (a+1))* which is *xfours ([2,4,5], (0 + 1))*

This matches the 3rd equation defining xfours, giving *xfours ([4,5], (0+1))*, which matches the 2nd equation and reduces to *xfours ([5], ((0 + 1) + 1))*. Notice how the second item of the 2-tuple is NOT evaluated yet – the patterns in the three equations defining xfours never require the second item to be evaluated, and so it isn't

The next reduction is to *xfours ([], ((0+1) +1))*, which matches the 1st equation defining xfours and reduces to *a*, which is *((0+1)+1)*

Finally, the + operators are evaluated using δ reduction: first to *(1+1)* and then to *2*

FUNCTIONAL PROGRAMMING
DESIGNING SIMPLE PROGRAMS

CURRYING

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*

The types of these functions are different :

f :: *num* –> *num* –> *num*
g :: (*num*, *num*) –> *num*

The type of f contains two function arrows, corresponding to the λx and the λy. Application is also different:

- *f* 3 4
- *g* (3, 4)

Currying

Functions that take more than one argument either

- collect arguments into a tuple, or
- use a "Curried" definition (named after Haskell B. Curry, though proposed earlier by Moses Schönfinkel), where a function of two arguments can be expressed as a function of one argument that returns a function to be applied to the second argument

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*

- in *f* the arguments are separated by spaces: in *g*, they appear inside a tuple
- compare *f* with the lambda expression λ*x* .E where E is (λ*y* .(*x* + *y*))

The types of these functions are different:

f ::*num*–>*num*–>*num*
g :: (*num*, *num*) –> *num*

The type of *f* contains two function arrows, corresponding to the λx and the λy. Application is also different, where *f* uses spaces whilst *g* uses a tuple:

- *f* 3 4
- *g* (3, 4)

We shall see that Currying is a very powerful concept

FUNCTIONAL PROGRAMMING DESIGNING SIMPLE PROGRAMS

CURRIED ACCUMULATIVE VERSION

```
fives :: [num] -> num
fives input
        = xfives input 0
          where
          xfives []          a = a
          xfives (5: rest)   a = xfives rest (a+1)
          xfives (x : rest)  a = xfives rest a
```

Curried accumulative version

For variety, we count how many times the number 5 appears in the input

This version is very similar to the *fours* function seen previously, but it uses a curried style for the definition of the local function – in this case *xfives*

xfives takes two data items, but instead of appearing in a 2-tuple they appear separately

Notice how the patterns used in the three equations defining *xfives* treat the two data items separately – they are not in an enclosing tuple pattern

Also notice that in each of the two recursive calls *xfives* is applied to two data items separated by a space (not collected together inside a 2-tuple)

FUNCTIONAL PROGRAMMING
DESIGNING SIMPLE PROGRAMS

```
fives :: [num]− > num
fives input
        = xfives input 0
          where
          xfives []          a = a
          xfives (5: rest)   a = xfives rest (a+1)
          xfives (x : rest)  a = xfives rest a

   fives [1,5,4,5]
-> xfives [1,5,4,5] 0
-> xfives [5,4,5] 0
-> xfives [4,5] (0 + 1)
-> xfives [5] (0 + 1)
-> xfives [] ((0 + 1) + 1)
-> ((0 + 1) + 1)
-> (1 + 1)
-> 2
```

Here we illustrate the evaluation of the *curried* accumulative recursive solution, when the function *fives* is applied to the list *[1,5,4,5]*

The first step is that *fives [1,5,4,5]* reduces to *xfives [1,5,4,5] 0*

This matches the 3rd equation defining xfives, so reduces to *xfives rest a* which is *xfives [5,4,5] 0*

This matches the 2nd equation, thus reducing to *xfives [4,5] (0 + 1)*

As with the previous accumulative recursive solution, notice that the pattern matching in the equations defining *xfives* do not require the second data item to be evaluated and so *(0 + 1)* is not evaluated until later

The remaining evaluation steps are firstly to *xfives [5] (0 + 1)* and then to *xfives [] ((0 + 1) + 1)* and then *((0 + 1) + 1)*

The final two steps are δ reductions, first to *(1 + 1)* and then to *2*

FUNCTIONAL PROGRAMMING DESIGNING SIMPLE PROGRAMS

PARTIAL APPLICATIONS

- Only Curried functions can be partially applied
- Example Curried definition:

```
f :: num –> num –> num -> num
f x y z = x + y + z
```

- Example full application:

```
main = f 3 4 5
```

- Example partial application bound to the name “g”:

```
g :: num -> num -> num
g = f 3
```

Partial applications

A great advantage of Curried functions is that, unlike uncurried definitions, they can be partially applied

For example, consider the Curried definition

```
f :: num –> num –> num -> num
f x y z = x + y + z
```

A “full application” of this function would look like this:

```
main = f 3 4 5
```

However, because it is Curried, the function *f* can be partially applied either to just its first argument or its first two arguments, and furthermore we can bind these partial applications to names (which will have function types):

```
g :: num -> num -> num
g = f 3

h :: num -> num
h = f 3 4
```

The name *g* is bound to a function that takes two arguments and returns their sum plus 3. It can be applied like this:

```
main = g 5 1
```

The name “*g*” gets replaced by (*f 3*) so this evaluates to *f 3 5 1* which is *9*

The name *h* is bound to a function that takes one argument and adds 7 to that argument. Partial applications can also be used without names (see later)

FUNCTIONAL PROGRAMMING
DESIGNING SIMPLE PROGRAMS

In summary, this lecture has explored approaches to designing simple functional programs. It started with a discussion of various solutions to the programming exercise set at the end of the last lecture, including stack recursive and applicative recursive styles. It has also covered Curried function definitions and partial applications

Summary

- Answers to exercise
  - Stack recursion answer
  - Accumulative recursion answer
- Currying
  - Curried accumulative recursion answer
- Partial applications

FUNCTIONAL PROGRAMMING
DESIGNING SIMPLE PROGRAMS