Functional Programming

Christopher D. Clack

FUNCTIONAL PROGRAMMING

Lecture 7

DESIGNING SIMPLE FUNCTIONS

continued

FUNCTIONAL PROGRAMMING
DESIGNING SIMPLE PROGRAMS

This lecture continues the discussion of partial applications that we started in the previous lecture – this time, in relation to the partial application of built-in operators

Two techniques for designing simple recursive functions are then explored: Case Analysis and Structural Induction

CONTENTS

- Partial applications of operators
- Case Analysis
- Structural Induction

FUNCTIONAL PROGRAMMING
DESIGNING SIMPLE PROGRAMS

PARTIAL APPLICATIONS OF OPERATORS

The built-in arithmetic operators have Curried definitions and can be partially applied

Miranda (+) ::
num –> num –> num

Miranda (+ 3) ::
num –> num

Operator presections:

$(3+)\ x \equiv (3 + x)$
$(5-)\ x \equiv (5 - x)$
$(2*)\ x \equiv (2 * x)$
$(9/)\ x \equiv (9/x)$

Operator postsections:

$(+3)\ x \equiv (x + 3)$
$(*2)\ x \equiv (x * 2)$
$(/9)\ x \equiv (x\ /9)$

To finish our discussion of partial application, we consider the partial applications of operators

The built-in infix operators have curried definitions and can be partially applied to just one argument, resulting in a function of one argument (these are always parenthesised, to avoid ambiguity):

Miranda (+) ::
num–> num–> num

Miranda (+ 3) ::
num –> num

Infix operators with two arguments can be partially applied either to their first argument (called a "presection") or to their second argument (called a "postsection"). **Post-sections are only possible for operators, not for functions!**

$(3+)\ x \equiv (3 + x)$
$(2*)\ x \equiv (2 * x)$
$(5-)\ x \equiv (5 - x)$
$(9/)\ x \equiv (9/x)$

$(+3)\ x \equiv (x + 3)$
$(*2)\ x \equiv (x * 2)$
$(/9)\ x \equiv (x\ /9)$

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

Later we will see examples of how to use operator sections

FUNCTIONAL PROGRAMMING
DESIGNING SIMPLE PROGRAMS

Approaches to design

1. Case Analysis
2. Structural Induction

Approaches to design

Here we will discuss two very simple approaches to designing functions – practical advice that can be helpful especially when tired and over-worked

1. Case Analysis (see the Miranda book, Section 3.7.1)
   - consider what VALUES can occur as input to a function
   - use Pattern Matching to match exact values, or use conditionals for relational tests
   - infer the general "looping" case of a recursive function from inspection of the base cases
2. Structural Induction (see the Miranda book, Section 3.7.2)
   - use the base case(s) and an induction hypothesis to help infer the "looping" part of a recursive function

FUNCTIONAL PROGRAMMING
DESIGNING SIMPLE PROGRAMS

Case Analysis example

```
myreverse :: [*] -> [*]
myreverse []              = []
myreverse (x:[])          = (x:[])
myreverse (x: (y:[]))     = (y: (x:[]))
myreverse (x:(y:(z:[]))) = (z:(y:(x:[])))
myreverse (x:xs)          = ?????
```

Look at the base cases and their solutions and try to find a common theme

```
myreverse (x:xs) = (myreverse xs) ++ [x]
```

Final version:

```
myreverse [] = []
myreverse (x:xs) = (myreverse xs) ++ [x]
```

Case Analysis example

Consider the function "myreverse", which takes a list of anything and returns the list with the order of elements reversed:

```
myreverse :: [*] -> [*]
myreverse []              = []
myreverse (x:[])          = (x:[])
myreverse (x: (y:[]))     = (y: (x:[]))
myreverse (x:(y:(z:[]))) = (z:(y:(x:[])))
myreverse (x:xs)          = ?????
```

To design the looping part of the function requires thought. Look at the base cases and their solutions and try to find a common theme. This common theme can often be used for the general "looping" equation for the function

In this case the theme is "reverse the rest of the list and put x at the end". And so we can write:

```
myreverse (x:xs) = (myreverse xs)++[x]
```

NB: x must be placed inside a list "[x]" because ++ expects two arguments both of which are lists. This is unlike cons (:) which expects an element and a list

FUNCTIONAL PROGRAMMING
DESIGNING SIMPLE PROGRAMS

Structural Induction

Structural induction is specifically used for recursive functions operating on lists, and works by considering the structure of the list

Structural Induction requires consideration of:

1. The base case that will terminate the recursion
2. The Induction Hypothesis (which is the assumption that the recursive function will work correctly for a list of a smaller size)
3. The Inductive Step – which is how to write the function body for the general (looping) equation for the function, given the Induction Hypothesis is true

Structural Induction

Induction infers a general rule from particular instances – for example, inferring the general "looping" equation for a recursive function by inspection of the base cases and the way that the function changes each time around the loop

Structural induction is specifically used for recursive functions operating on lists, and works by considering the structure of the list

Structural Induction requires consideration of:

1. The base case that will terminate the recursion (sometimes there may be more than one base case)
2. The Induction Hypothesis (which is the assumption that the recursive function will work correctly for a list of a smaller size – even though you have not yet written the function!)
3. The Inductive Step – which is how to write the function body for the general (looping) equation for the function, given the Induction Hypothesis is true (this is the only bit that requires creative thought)

FUNCTIONAL PROGRAMMING
DESIGNING SIMPLE PROGRAMS

Induction on the structure of a list

…

(x:(y:(z:[]))) has three elements

(x:(y:[])) has two elements

(x:[]) has one element

[] has zero elements

The looping part of a recursive function on a list makes a recursive call to a lower rung on the ladder (P.O.R. list gets smaller)

Induction Hypothesis: recursive call on lower rung works!

Inductive Step: expression using the recursive call

Finally solve the base case

When we say that Structural Induction is based on the structure of a list we mean how many elements a list has. Consider a "ladder" of all possible lengths that a list might have:

…

(x:(y:(z:[]))) has three elements

(x:(y:[])) has two elements

(x:[]) has one element

[] has zero elements

Many recursive functions on lists use [] as the base case for recursion. Also, each rung of the ladder has a rung above it. When designing a recursive function that takes a list as input, the general "looping" equation defining the function will make a recursive call to a smaller list – typically one element smaller. This corresponds to one rung down on the ladder of the list that is the parameter of recursion (that's the list argument that MUST change on each loop, eventually reaching the base case).

The "trick" of structural induction is to find a way to use the recursive call so that ASSUMING it gives the correct result for one rung on the ladder then we are guaranteed to get the correct result for the next rung up on the ladder. This assumption is the Inductive Hypothesis, and the expression using the recursive call is the Inductive Step

If you now solve the function for the base case it's guaranteed to work for all cases

FUNCTIONAL PROGRAMMING
DESIGNING SIMPLE PROGRAMS

Structural Induction Example

The function “startswith” takes a 2-tuple containing two lists of anything and returns True if the second list starts with the first list. Otherwise it returns False

startswith ([1,2], [1,2,3,4]) returns True

startswith ([1,2], [2,3,4]) returns False

Three steps:

1. Specify the TYPE

2. Consider the General Case (the looping case)
   - Identify the parameter of recursion
   - State the Induction Hypothesis
   - Find the Inductive Step

3. Consider the base case(s)

Structural Induction example

The function “startswith” takes a 2-tuple containing two lists of anything and returns True if the second list starts with the first list. Otherwise it returns False

For example:

startswith ([1,2], [1,2,3,4]) gives True

startswith ([1,2], [2,3,4]) gives False

Designing the function *startswith* requires three steps:

1. Specify the TYPE
2. Consider the General Case (the looping case)
   - Identify the parameter of recursion
   - State the Induction Hypothesis
   - Find the Inductive Step
3. Consider the base case(s)

FUNCTIONAL PROGRAMMING
DESIGNING SIMPLE PROGRAMS

Worked example:

1. *startswith :: ([*], [*]) -> bool*

2. Consider the possible recursive calls:

startswith (xs, (y:ys))	only the first list changes
startswith ((x:xs), ys)	only the second list changes
startswith (xs, ys)	both lists change

Which of the above recursive calls best helps us to define *startswith ((x:xs), (y:ys))* ?

Here is a worked example of the procedure for designing the *startswith* function

1. The type is straightforward:

   ([*], [*]) -> bool

2. The most general equation to define this function will use general list patterns for both of the two argument lists:

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

We must (i) identify the parameter of recursion ("PoR"), (ii) state the induction hypothesis, and (iii) find the inductive step. The first two of these often happen at the same time because (ii) will be the assumed correct operation of a recursive call where (i) is one element smaller. For *startswith*, the possible recursive calls are:

startswith (xs, (y:ys))	xs is the PoR
startswith ((x:xs), ys)	ys is the PoR
startswith (xs, ys)	xs, ys are both PoR

Which recursive call best helps us to define *startswith ((x:xs), (y:ys))* ?

FUNCTIONAL PROGRAMMING
DESIGNING SIMPLE PROGRAMS

Worked example (continued)

Answer: *startswith (xs, ys)* is the best choice of recursive call because it allows us to write:

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

(i) Induction Hypothesis: *startswith (xs, ys)* works correctly

(ii) Parameter of Recursion: both lists are parameters of recursion

(iii) Inductive Step: the result is True only if the head of each list is the same and the recursive call returns True

Simpler version:

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

Answer: *startswith (xs, ys)* is the best choice of recursive call because it allows us to write:

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

Here we have done three things at once:

(i) Identified the Induction Hypothesis, which is that the recursive call *startswith (xs, ys)* works correctly

(ii) Identified the parameter of recursion – in this example, both lists are parameters of recursion (both change on every loop)

(iii) Found the Inductive Step (i.e. given the Induction Hypothesis, how to write the code so that the general loop works correctly)

A simpler version of this solution is:

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

FUNCTIONAL PROGRAMMING
DESIGNING SIMPLE PROGRAMS

Worked example (continued)

Base case(s):

startswith ([], any) = True
startswith (any, []) = False

And the final solution is:

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

3. Finally we must consider the base case(s)

Because there are two parameters of recursion, we must consider two base cases that terminate the recursion. This arises where either the first list or the second list is empty:

(i) startswith ([], any)

(ii) startswith (any, [])

For the former, there is no obvious right or wrong solution – we choose to return True (i.e. that any list starts with nothing). For the latter, the result is obvious since any empty list cannot start with something. Therefore we have:

startswith ([], any) = True
startswith (any, []) = False

And the final solution is:

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

FUNCTIONAL PROGRAMMING
DESIGNING SIMPLE PROGRAMS

In summary, this lecture has briefly continued our previous discussion of partial application and then introduced two simple approaches to function design that can help when the solution to the function appears difficult – the first was Case Analysis, and the second was Structural Induction (for use with functions over lists)

Summary

- Partial applications of operators
- Case Analysis
- Structural Induction

FUNCTIONAL PROGRAMMING
DESIGNING SIMPLE PROGRAMS