Functional Programming

Christopher D. Clack

FUNCTIONAL PROGRAMMING

Lecture 8

RECURSION AND THE λ CALCULUS

FUNCTIONAL PROGRAMMING RECURSION AND THE λ CALCULUS

This lecture jumps back to consider the Lambda Calculus and how recursion can be encoded. It compares simple Miranda functions with their equivalents in the λ calculus, explains the problem of name binding, and introduces the fixed-point operator "Y". Finally a λ calculus definition for "Y" is given using self application

CONTENTS

- Introduction
- Recursive functions in Miranda
- Recursive functions in the *λ* calculus
  - Binding for a function name
  - Fixed point
  - Fixed point operator
  - Self-application

FUNCTIONAL PROGRAMMING RECURSION AND THE λ CALCULUS

Overall structure of understanding how to define a recursive function in the λ calculus:

- Given a recursive function “f”
- Create a non-recursive function “h”
- Knowledge of “fixed points” and the fixpoint operator “Y”
- Create a definition of “f” in terms of ”h” and “Y”
- Express “Y” in the λ calculus using self-application
- Give a final λ calculus definition of “f” without “Y”

Recursion and the λ calculus

The previous two lectures have given examples of recursive functions in Miranda, but in our earlier discussion of the λ calculus we did not mention how to define recursive functions

This lecture will start by comparing non-recursive Miranda functions with their λ calculus equivalents, and highlight the problem of name-binding when we try to define a recursive function in the λ calculus

The route to a solution will be as follows:

- State a required recursive function (call it “f”) as a Miranda program and then as a (not yet working) λ calculus expression
- Create a related non-recursive function “h” – both in Miranda and the λ calculus
- Explain a little about “fixed points” and the fixpoint operator “Y”
- Create a definition of “f” in terms of ”h” and “Y”
- Explain how the operator “Y” can be expressed in the λ calculus using self-application
- Give a final λ calculus definition of “f” without “Y” that works correctly

FUNCTIONAL PROGRAMMING RECURSION AND THE λ CALCULUS

A non-recursive Miranda function:

f x = 3, if (x=0)

= 11, otherwise

In our extended λ calculus this is:

λ x.(if (x = 0) 3 11)

A recursive Miranda function:

f x = 3, if (x=0)
= 1 + f (x-1), otherwise

What's wrong with this? :-

λ x.(if (x = 0) 3 (1 + (f (x-1))))

Consider the following non-recursive Miranda function:

f x = 3, if (x=0)

= 11, otherwise

In our extended λ calculus (with added constants and operators) this is:

λ x.(if (x = 0) 3 11)

This works correctly

Now consider the following recursive Miranda function:

f x = 3, if (x=0)

= 1 + f (x-1), otherwise

And what's wrong with the following λ calculus version of the recursive function?

λ x.(if (x = 0) 3 (1 + (f (x-1))))

[pause to think!]

The answer is that the name "f" is not bound to anything!

This is the core of the problem. A recursive function makes a call to itself inside its own function body, but in the λ calculus functions are anonymous – they have no names – and so how can a λ calculus function call itself?

FUNCTIONAL PROGRAMMING RECURSION AND THE λ CALCULUS

Consider a whole program:

f x = 3, if (x=0)
= 1 + f (x-1), otherwise
main = f 7

This translates to:

λf.(f 7) (λx.(if(x=0)3(1+(f(x-1)))))

Reductions:

(λx.(if (x = 0) 3 (1 + (f(x - 1))))) 7
(if (7 = 0) 3 (1 + (f(x - 1))))
(if False 3 (1 + (f(x - 1))))
(1 + (f(x - 1)))

There is no value bound to the name “f” so there is no way to evaluate “f (x-1)”

Perhaps the binding for “f” happens elsewhere?

Consider not just one function definition but a whole program. Here’s a simple example using the same Miranda function embedded in a program:

f x = 3, if (x=0)
= 1 + f (x-1), otherwise
main = f 7

This translates to the following λ calculus expression, which still has a problem with name binding:

λf.(f 7) (λx.(if(x=0)3(1+(f(x-1)))))

After the first β reduction the name binding problem persists:

(λx.(if (x = 0) 3 (1 + (f(x - 1))))) 7

And after a second β reduction:

(if (7 = 0) 3 (1 + (f(x - 1))))

The δ reduction for (7=0) doesn’t help:

(if False 3 (1 + (f(x - 1))))

Nor does the δ reduction of “if”:

(1 + (f(x - 1)))

There is still no value bound to the name “f” so there is no way to evaluate “f (x-1)”

FUNCTIONAL PROGRAMMING RECURSION AND THE λ CALCULUS

Define a non-recursive function "h" that is related to "f":

h f x = 3, if (x=0)

= 1 + (f (x-1)), otherwise

The function body is identical to the target function "f", and the λ calculus definition for "h" is:

λ f.(λ x.(if (x=0) 3 (1+(f(x-1)))))

Notice that: (h f) ≡ f

We say that "f is a fixed point of h" and we can use the operator "Y" to give us the fixpoint of "h"

So how can we define a recursive function "f" in the λ calculus?

First, we define a new NON-RECURSIVE Curried function ("h") that is related to the target function "f"

- Its body is identical to the body of "f"
- But it takes an additional argument which has the name "f"

h f x = 3, if (x=0)

= 1 + (f (x-1)), otherwise

Because "h" is not recursive, we can define a perfectly correct λ calculus expression for "h":

λ f.(λ x.(if (x=0) 3 (1+(f(x-1)))))

Of course "h" is not "f" so we haven't solved the problem yet. However, notice that the partial application "(h f)" behaves in exactly the same way as "f" and so we have an identity:

(h f) ≡ f

(NB: this is an identity, not a definition)

This identity gives us a way forward towards a λ calculus definition for the recursive function "f", because it says that "f" is a "fixed point" (or "fixpoint" of "h" and we can use an operator called "Y" to give us the fixpoint of "h" – this is explained in the following slides

FUNCTIONAL PROGRAMMING RECURSION AND THE λ CALCULUS

Fixed points

x is a fixed point ("fixpoint") of g if and only if g x = x

Every function in the λK calculus has a fixpoint

A special "fixpoint operator" called Y gives the fixpoint of any function in the λK calculus. Y takes a function (call it "g") as its argument and returns a fixpoint of that function "g".

Thus, by definition: g (Y g) ≡ (Y g)

And the δ rule for Y is: Y g = g (Y g)

We know from the previous slide that "f" is a fixpoint of "h", so we have:

f = Y h

Fixed points

In mathematics, a **fixed point** (or **fixpoint**) of a function is a value in the function's domain that is mapped to itself by the function. If x is a fixed point of a function g then x = (g x) = g (g x) = g (g (g x)) and so on

In general, a function may have many fixed points, as illustrated by the functions three and identity:

three x = 3 (one fixpoint)

identity x = x (every x is a fixpoint)

A fundamental theorem of the untyped λK calculus is that every function has a fixpoint (in the context of this discussion, we'll say that a λK calculus expression can be derived to represent the fixpoint of any function in the λK calculus, without worrying too much about what it *means*)

There is a special fixpoint operator (called "Y") that we can incorporate into the λ calculus and which will return the fixed point of any function. Y takes a function (call it "g") as its argument and returns a fixed point of "g". Thus, by definition (Y g) is a fixed point of g and we have the identity g (Y g) ≡ (Y g) and the δ rule:

Y g → g (Y g)

Returning to our need to obtain a λ calculus definition for the function "f", we know that "f" is a fixpoint of "h", so we have:

f = Y h

FUNCTIONAL PROGRAMMING RECURSION AND THE λ CALCULUS

Example Normal Order Reduction of (f 1)

f 1 → (Y h) 1	f = Y h
→ (h (Y h)) 1	δ rule for Y
→ *(λ f.(λ x.(if (x=0) 3 (1+(f(x-1))))) (Y h)) 1*	def of h
→ *(λ x.(if (x=0) 3 (1+((Y h)(x-1))))) 1*	β rule
→ *(if (1=0) 3 (1+((Y h)(1-1))))*	β rule
→ *(if False 3 (1+((Y h)(1-1))))*	δ rule for =
→ *1+((Y h)(1-1))*	δ rule for if
→ *1+((h (Y h))(1-1))*	δ rule for Y
→ *1+((λ f.(λ x.(if (x=0) 3 (1+(f(x-1))))) (Y h))(1-1))*	
→ *1+((λ x.(if (x=0) 3 (1+((Y h)(x-1))))(1-1))*	
→ *1+((if ((1-1)=0) 3 (1+((Y h)((1-1)-1))))*	
→ *1+((if (0=0) 3 (1+((Y h)((1-1)-1))))*	
→ *1+((if True 3 (1+((Y h)((1-1)-1))))*	
→ *1+3*	

Example evaluation of "f 1"

This slide provides a worked step-by-step example of the reduction steps to evaluate the function application (f 1) when expressed in the λ calculus

For each step of the first loop the relevant operation is given in black text on the right hand side. The explanations for the second loop can be derived from those given for the first loop

This demonstrates how the recursive δ rule for Y (which is that Y g → g (Y g)) creates the loop and there is no problem with name binding

FUNCTIONAL PROGRAMMING RECURSION AND THE λ CALCULUS

A λ calculus expression that defines "f":

Y (λ f.(λ x.(if (x = 0) 3 (1 + (f (x - 1))))))

But this depends on extending the λK calculus with the fixpoint operator "Y"

Can "Y" be defined using the λK calculus?

Yes: using self-application

e.g. infinite loop:

(λx.(x x)) (λx. (x x))

β→ (λx.(x x)) (λx. (x x))

Define Y as:

λq.((λx.(q (x x))) (λx.(q (x x))))

Now we have a λ calculus expression that defines "f":

Y (λ f.(λ x.(if (x = 0) 3 (1 + (f (x - 1))))))

However, this is not a λK calculus expression, it is λK calculus extended with constants and operators – and most especially the fixpoint operator "Y"

Is it possible to define the fixpoint operator "Y" just using the λK calculus?

The answer is "yes", using self-application!

In the λK calculus it is possible to write expressions that loop. For example the following expression loops forever if you attempt to evaluate it with β reduction:

(λx.(x x)) (λx. (x x))

β→ (λx.(x x)) (λx. (x x))

The above expression loops because x is applied to itself in the function body (in the sub-expression "(x x)")

We can define "Y" In the λK calculus as:

λq.((λx.(q (x x))) (λx.(q (x x))))

FUNCTIONAL PROGRAMMING RECURSION AND THE λ CALCULUS

Example showing how ”Y” works:

Y h

≡ *λq.((λx.(q (x x))) (λx.(q (x x)))) h* (line 1)

β→ *(λx.(h (x x))) (λx.(h (x x)))* (line 2)

β→ h ((λx.(h (x x))) (λx.(h (x x)))) (line 3)

(line 3) ≡ h (line 2)

(line 3) ≡ Y h

(line 2) ≡ Y h

Therefore:

Y h ≡ h (Y h)

Example showing how the λK calculus definition for Y works:

Y h

≡ *λq.((λx.(q (x x))) (λx.(q (x x)))) h* (line 1)

β→ *(λx.(h (x x))) (λx.(h (x x)))* (line 2)

β→ h ((λx.(h (x x))) (λx.(h (x x)))) (line 3)

Notice that line 3 ≡ h (line 2) and that line 3 = Y h and line 2 ≡ Y h

Therefore:

Y h ≡ h (Y h)

This is exactly the behaviour we require from “Y”

FUNCTIONAL PROGRAMMING RECURSION AND THE λ CALCULUS

Deriving a λK calculus definition for "f":

f ≡ (Y h)

≡ (Y (λ f.(λ x.(if (x=0) 3 (1+(f(x-1)))))))

≡ ((λq.((λx.(q (x x))) (λx.(q (x x)))))
(λ f.(λ x.(if (x=0) 3 (1+(f(x-1)))))))

Putting it all together we can derive a λK calculus definition for "f" as follows:

f ≡ (Y h)

Replace "h" with its λK calculus definition to give:

≡ (Y (λ f.(λ x.(if (x=0) 3 (1+(f(x-1)))))))

And replace "Y" with its λK calculus definition to give:

≡ ((λq.((λx.(q (x x))) (λx.(q (x x)))))
(λ f.(λ x.(if (x=0) 3 (1+(f(x-1)))))))

FUNCTIONAL PROGRAMMING RECURSION AND THE λ CALCULUS

This lecture has explored how recursion can be encoded in the λ calculus. It has compared simple Miranda functions with their equivalents in the λ calculus, explained the problem of name binding, and introduced the fixed-point operator "Y". Finally a λ calculus definition for "Y" was given using self application

SUMMARY

- Introduction
- Recursive functions in Miranda
- Recursive functions in the *λ* calculus
  - Binding for a function name
  - Fixed point
  - Fixed point operator
  - Self-application