What is functional programming?

Craig Aspinall
Co-host of Coding By Numbers

Today I'm going to talk about functional programming. I started looking at this last year because I was keen to understand what all the hype was about. From the little bit I did know then, I also thought it would enable me to write better, more testable, more maintainable code.

One year on I would say that I know enough about it to know that I still have a long long way to go, but I'm already writing better code as a result.

Unlike previous talks I've given here here at Coding By Numbers, I hope that this one will give you something practical you can take away and apply to the code you write tomorrow. Or at least that it will make you think about how you write your code tomorrow.

The stuff of myth and legend!

There is a lot of misinformation and misinterpretation around what functional programming is.

It’s not the black art that some would have you believe!

Let us start by defining some of the things that functional programming is not.

Let’s start by dispelling some of the myths. Functional programming is still seen as a black art by some developers, something occult, which of course, it isn’t.

It does inherit a lot of terminology from branches of advanced mathematics, such as category theory, but you don’t need to know or understand that to get started with functional programming. At it’s very essence, it’s quite simple so let’s clarify some of the things that functional programming is not about.

It is not about…

- Type systems:
  - strong versus weak
  - static versus dynamic
- Lazy versus strict evaluation
- Higher order functions
- Monads, monoids, functors etc.

These questions apply to both functional and non-functional programming!

It is not about type systems. The questions of strong versus weak, and dynamic versus static typing have nothing to do with functional programming. Neither does the question of lazy versus strict (or eager) evaluation.

There are lots of features that are commonly associated with functional programming like higher order functions, monads, monoids, functors etc. but functional programming is not about those either.

All of these things can apply to both functional and non functional programming.

So what is functional programming about?

To answer that, we need to go back in time, to the 1930s.

So what is functional programming really about? To answer that question we need to look back to where it all began, in 1930s America.

This was of course the time of the great depression. Unemployment was rife and many families were struggling to put food on the table. Even the wealthy had lost a fortune on the stock market and were forced to sell their belongings at knocked down prices.

Princeton University

In the 1930s at Princeton university, there were a number of individuals trying to answer questions related to **computation**, such as:

> "If we had infinite computing power, what problems would we be able to solve?"

With industry and society on its knees, many of the best minds of the time worked in academia. The top universities were a haven from what was going on in the rest of the country.

From a functional programming perspective, we're most interested in Princeton University in New Jersey. There were a number of individuals there who were interested in the subject of computation, and were trying to answer questions such as:

- If we had infinite computing power, what problems would we be able to solve?
- Would there be any problems we could not solve?
- Could they be solved in more than one way?

If you come from a computer science background, you may recognize the names of a few of them!

Alonzo Church

Alonzo Church developed a formal system called **lambda calculus**, essentially a programming language for a machine with infinite power.

It was based on **functions** that took other **functions** as parameters and returned **functions**.

Using this he was able to provide conclusive answers to the many of the computation questions.

One of those individuals was Alonzo Church. He developed a formal system called lambda calculus that was essentially a programming language for one of these imaginary machines with unlimited computing power.

Lambda calculus is based on functions that take other functions as parameters and return functions as results. You can probably guess where we're going with this!

Using lambda calculus, Church was able to provide answers to many of the computation related questions that the academics were asking.

Alan Turing

Alan Turing independently developed a different formal system now known as the **Turing machine**.

Using his system Turing arrived at similar conclusions to Alonzo Church.

It was later shown that Turing machines and lambda calculus were equivalent in power.

Another one of the individuals at Princeton was Alan Turing. If you haven't heard of Alonzo Church before I'm sure you will have heard of the Turing machine.

The Turing machine was another formal system that Turing developed independently of the work that Church was doing, and using it he was also able to provide answers to the computation related questions.

Unlike lambda calculus, a Turing machine is very much a state based system, reading instructions from a tape and performing actions such as moving the tape head back and forth and writing output. The concept of a function is missing from it.

It was later shown that Turing machines and lambda calculus were equivalent in power. Whatever could be achieved in one could also be achieved in the other. This is important to remember as we look at how we can perform functional programming in different programming languages.

John von Neumann

John von Neumann developed what we now call “von Neumanns architecture”.

In 1949 the EDVAC (Electronic Discrete Variable Automatic Computer) was unveiled and was the first example of von Neumanns architecture, effectively a real world implementation of a Turing machine.

Another person who was at Princeton at the time and you might have heard of, was John von Neumann. He developed what we now call “von Neumann’s architecture”, something that is still prevalent in todays CPUs.

In 1949 the EDVAC (Electronic Discrete Variable Automatic Computer) was unveiled and was the first example of von Neumanns architecture, effectively a real world implementation of a Turing machine. So this time Church missed out and Turing claimed the crown, but Church was still to have his day.

John McCarthy

In 1958, John McCarthy unveiled the LISP (LISt Processing) language whilst working at MIT.

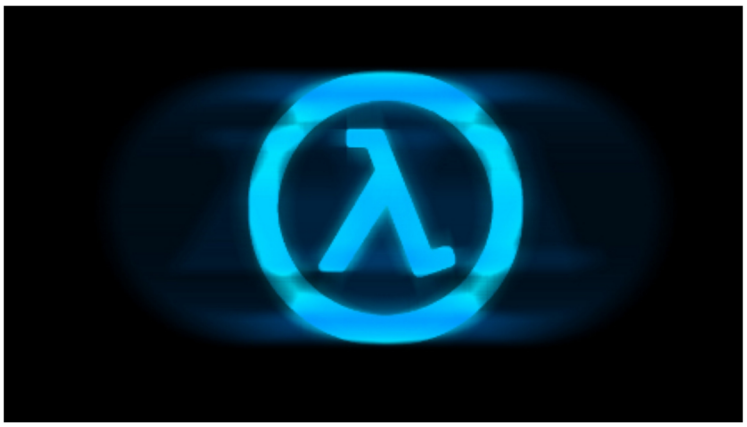
This was an implementation of lambda calculus that worked on von Neumann architectures.

Almost 10 years later, in 1958, John McCarthy (a Princeton graduate working at MIT at the time) unveiled LISP (LISt Processing language). LISP was an implementation of lambda calculus that worked on von Neumann architectures. This proved that the Turing machine and lambda calculus were equivalent and one could be implemented in terms of the other.

There are still ardent supporters of LISP who believe that it is the one true way of programming computers, and yet it still hasn’t captured the mindshare of the majority of developers out there.

So, what is functional programming?

Unsurprisingly, functional programming is a practical implementation of lambda calculus and at the core of it is the **function**!



> Practical in the sense that not all lambda calculus ideas can be implemented under physical limitations!

Hopefully you arrived at the answer by now, but functional programming is a practical implementation of Church's lambda calculus. We say practical because lambda calculus was intended for machines with infinite computing power, which we don't have (yet), so not all of the ideas translate to reality.

And the most important part of lambda calculus for practical functional programming is the function itself. Remember that functions were used for nearly everything in lambda calculus. So we should look at what a function is and where variables come in?

So, what is functional programming?

A variable:

- is an alias for an expression
- is assigned to once
- is never modified

Well, in functional programming variables are really just aliases for expressions. We can assign to them once and then we can’t modify them.

If we want to mutate a variable then we create a copy with the new value and leave the old one in place. Sometimes this is enforced by the language and the rest of the time we have to be disciplined about it.

So, what is functional programming?

A function:

- always takes at least one argument
- always returns a result
- does not operate on anything other than its arguments (if it's pure)
  - no side effects
  - guaranteed to return the same result given the same arguments

---

What are functions?

Functions always take at least one argument and return some result. They do not operate on anything other than their arguments, at least not if they're pure functions, which is what we are aiming for. Pure functions have no side effects, and by side effects we mean that the only visible change in the system as a result of calling the function is whatever it returns. Pure functions are therefore guaranteed to return the same result given the same argument values.

Note that there can be no such thing as a void function because that would imply that either:

1. the function doesn't do anything, in which case it is pointless, or
2. it is mutating one of the arguments which we supplied, which means we don't have immutable data and we can't guarantee side effect free code, or
3. it's operating on something other than the arguments which we supplied, which also means we can't guarantee that it is free from side effects!

Why should we care?

"referential transparency"

So at this point you might be thinking that this functional programming lark doesn't sound quite as scary as you thought it was. You're also possibly thinking that you and the majority of developers in the last couple of decades have been managing very nicely with imperative object oriented programming thank you very much, so why would you be bothered to fix something that isn't broken?

Well, it all boils down to referential transparency, which is a fancy way of saying the last point on the previous slide. Given a pure function and a set of immutable argument values, we will always get the same result, regardless of when we call it, or how many times we call it, or what other functions we call before it. It guaranteed to return the same result every time.

It's predictable. And that has major implications.

Referential transparency means…

- easier testing
- easier debugging
- easier re-use
- implicit thread safety

Referentially transparency makes it much easier to unit test your code, because you have well defined inputs and outputs to your functions, and the only thing that can affect those output are the inputs.

Debugging your code becomes much simpler too, because the behaviour of your function does not depend on anything that happened before it was called, it only depends on the values of its arguments.

Since your functions don't depend on anything around them, and they tend to have a single, well defined purpose, they are easier to re-use too.

It also means that your code is implicitly thread safe, because there is no shared mutable state, so it is ready to take advantage of all those CPU cores!

Whilst there is no shared mutable state, it isn't true to say that there is no state at all. It's just that the state is held on the stack in the function arguments, rather than a shared location in the heap.

Don’t I need to be using a functional language?

So… do you need to switch to a functional programming language to do functional programming? Well, I hate to disappoint you but the whole functional versus imperative programming argument is really a false dichotomy.

In fact you can use any of these languages to do functional programming, and that makes sense when you think back to the equivalence of lambda calculus and the Turing machine. Anything you can do in one, you can also do in the other.

It is true to say however, that certain languages lend themselves better to functional programming than others, and the languages on the right, Haskell, Erlang, Clojure and other LISP implementations, lend themselves better to functional programming than the languages on the left, Java, C#, Ruby and Python, with the languages in the middle, Scala, OCaml and F# being somewhere in between.

Language traits that aide functional programming

- Declarative – describe what rather than how
- Functions are first class values – they can be passed around
- Higher order functions – functions that operate on other functions
- Immutable data – cannot modify data once initialized
- Recursive loops – instead of iteration (for, while, repeat etc.)
- Algebraic data types – simple containers that can be recursively defined
- Pattern matching – on primitive values and algebraic data types

---

Declarative languages tend to lend themselves better to functional programming because they describe the result they want rather than the steps required to obtain the result.

Functions as first class values and higher order functions make it easier to pass around functions which makes them easier to re-use.

Immutable data helps enforce referential integrity and save us from ourselves, at least some of the time.

Loops are handled recursively rather than iteratively.

Algebraic data types also bring recursion into type definitions.

And pattern matching allows you to return results based on values or structure without the need for buckets of if-then-elses statements.

FizzBuzz

This is a simple coding exercise I used to get candidates to complete in five minutes during telephone interviews using Google Docs.

Write a program that prints the numbers from 1 to 100 except:

- for each number that is a multiple of 3, print "Fizz" instead of the number
- for each number that is a multilpe of 5, print "Buzz" instead of the number
- for each number that is a multiple of 3 and 5, print "FizzBuzz" instead of the number

---

So let's look at a real example. This is an exercise I used to get job candidates to do during technical telephone interviews for Java and C++ roles. We just did in Google Docs and I didn't try to compile the answer they gave. I just wanted to see that they could complete a simple programming task and I was mostly interested in seeing that they understood iteration and control flow, and not worried about missing semi-colons or anything like that.

The exercise is really simple. Print the numbers 1 to 100 but for every multiple of 3 print "fizz", for every multiple of 5 print "buzz" and for every multiple of 3 and 5 print "fizzbuzz".

You would be amazed at the number of people who have allegedly been to university studying computer science, and/or working as a developer for years, who could not complete this exercise satisfactorily!

Typical “FizzBuzz” Answer in Java

```
public class FizzBuzz {
        public static void main(String[] args) {
                for (int i = 1; i <= 100; i++)
                        if ((i % 3 == 0) && (i % 5 ==0))
                                System.out.println("FizzBuzz");
                        else if (i % 3 == 0)
                                System.out.println("Fizz");
                        else if (i % 5 == 0)
                                System.out.println("Buzz");
                        else
                                System.out.println(i);
        }
}
```

This is pretty much what I would expect most Java developers to write. It’s not very pretty but does what it’s supposed to and demonstrates the basic concepts I was looking for.

It’s also very dense and difficult to work out from the code what it is supposed to be doing. If I hadn’t already explained the exercise to you, it would probably take you a minute or two to figure it out.

Our IO is intertwined with our “business logic” and it would be very difficult to test as a result. We would have to execute the main() method and capture the output from stdout to be able to prove anything.

A Better “FizzBuzz” Answer

```
import java.util.*;

public class BetterFizzBuzz {
        static List<String> fizzBuzz(int upperBound) {
                ArrayList<String> results = new ArrayList<String>();
                for (int i = 1; i <= upperBound; i++)
                        if ((i % 3 == 0) && (i % 5 ==0))
                                results.add("FizzBuzz");
                        else if (i % 3 == 0)
                                results.add("Fizz");
                        else if (i % 5 == 0)
                                results.add("Buzz");
                        else
                                results.add(String.valueOf(i));
                return results;
        }

        public static void main(String[] args) {
                for(String line: fizzBuzz(100))
                        System.out.println(line);
        }
}
```

Here I’ve refactored the code to make it more testable.

I’ve separated the IO from the business logic, by creating a function that accepts an int and returns a List of Strings. The integer argument tells us which number we want to “fizzbuzz” up to, and the list of strings is the corresponding “fizzbuzz” sequence.

And as you can see, now the main method just iterates over what is returned from our fizzBuzz() function and outputs it. This part of the code is so simple it shouldn’t require any specific testing.

Since the fizzBuzz() function doesn’t operate on anything other than it’s arguments we can test it thoroughly in solation of the IO code. Once we have some tests in place, then we can look at further refactorings.

A More Functional “Fizzbuzz” Answer in Java

```
import java.util.*;

public class FunctionalFizzBuzz {
        static boolean isMultipleOf3(final int num) { return num % 3 == 0; }

        static boolean isMultipleOf5(final int num) { return num % 5 == 0; }

        static String fizzBuzzOrNumber(final int num) {
                if (isMultipleOf3(num) && isMultipleOf5(num))
                        return "FizzBuzz";
                else if (isMultipleOf3(num))
                        return "Fizz";
                else if (isMultipleOf5(num))
                        return "Buzz";
                else
                        return String.valueOf(num);
        }

        static List<String> fizzBuzz(final int upperBound) {
                ArrayList<String> results = new ArrayList<String>();
                for (int i = 1; i <= upperBound; i++)
                        results.add(fizzBuzzOrNumber(i));
                return Collections.unmodifiableList(results);
        }

        public static void main(String[] args) {
                for (String line: fizzBuzz(100))
                        System.out.println(line);
        }
}
```

This appears to be much longer but it is also much easier to figure out what it is supposed to be doing when you don’t have any other information (as is the case with most legacy code I’ve ever come across).

1. The main() method is now iterating over the collection of Strings returned from the fizzBuzz() methods and printing them out.
2. The fizzBuzz() method is counting from 1 to the supplied upper bound and adding the result of fizzBuzzOrNumber() to a list of Strings, which it then returns.
3. The fizzBuzzOrNumber() method checks to see if the number it is passed is a multiple of 3 or 5, both or neither, and returns different results accordingly.
4. We probably don’t even need to look at the implementations of isMultipleOf3() or isMultipleOf5() to know what they do because it’s pretty obvious from the name and the context in which they are used.

We now have something we can now thoroughly test. All of the “methods” are actually pure functions, with a single, clear purpose, that operate only on their arguments. I’ve also made all the variables immutable so the functions are referentially transparent too.

A Functional “FizzBuzz” Answer in Haskell

Note that I wrote this, so it may not be the most idiomatic Haskell example you will find!

```
import Control.Monad

multipleOf3 num = num `mod` 3 == 0

multipleOf5 num = num `mod` 5 == 0

fizzBuzzOrNumber num =
        if multipleOf3 num && multipleOf5 num then "FizzBuzz" else
                if multipleOf3 num then "Fizz" else
                        if multipleOf5 num then "Buzz"
                                else show num

fizzBuzz upperBound = map fizzBuzzOrNumber [1..upperBound]

main = forM_ (fizzBuzz 100) putStrLn
```

This is an approximation of the functional “FizzBuzz” answer in Java and probably doesn’t represent idiomatic Haskell, but it makes it easier to point out where there are significant differences.

Haskell uses type inference so we don’t have to declare any types, although everything is strongly and statically typed.

It doesn’t require parentheses around function arguments or semi-colons at the end of functions which reduces a lot of the noise from Java and other C derived languages.

We can place binary functions in the infix position simply by surrounding the function name with back quotes.

if-then-else is not used for control flow as it is in imperative languages like Java, it is used to select from two possible return expressions based on some boolean expression. In this case we have nested if constructs but they all ultimately resolve to return a single value or expression.

Whitespace is used to distinguish new declarations from line continuations. If we placed any of the if constructs at the left margin it wouldn’t compile.

The map function takes two arguments, a function to convert values of type A to values of type B, and a list of values of type A, and it returns a list of values of type B. In this case we pass it a list of integers from 1 to the upper bound, and the fizzBuzzOrNumber function to convert the integer values into the corresponding strings.

Finally we have the main function. Here the forM_ function accepts a list of values of type A and a function that operates on values of type A and it applies the function to every value in the list in order. We supply the result of (fizzBuzz 100) as the list and putStrLn which outputs strings to stdout followed by a newline character.

You may or may not have noticed a number a significant language traits in

action here.

1. fizzBuzz and forM_ are higher order functions because they accept other functions as arguments, which in turn means that functions must be first class values
2. Haskell lists are a recursively defined algebraic data type
3. There are 2 instances of monads on show:
    1. Haskell lists are also instances of monads
    2. All IO in Haskell is contained within the IO monad

Hopefully, this example isn't that scary or difficult, even though there are a number of language features being used that you may or may not be used to.

Learning Haskell

- haskell.org
- Learn you a Haskell – learnyouahaskell.com
- Real World Haskell – book.realworldhaskell.org

If that has whet your appetite for learning Haskell, then the best place to start is at the haskell.org website where you can download the Haskell platform and find lots of information about it.

In terms of learning Haskell I would stringly recommend Learn You A Haskell. The entire book has just been published in paper and e-book form but all the content is available free at the website.

Real World Haskell is also a good resource that tries to teach Haskell through a number of "real world" case studies.

And last but not least, get yourself down to the Brisbane functional programming group this Tuesday where Tony Morris is doing a second hands on session with Haskell. That will be one of your best opportunities to bootstrap your learning.

More information about functional programming

- http://aspinall.github.com/talks/what-is-fp/what-is-fp.html
- Functional Programming For The Rest of Us – Slava Akhmechet at defmacro.org
- What Does Functional Programming Mean? – Tony Morris at BFPG
- Introduction to FP – Brad Clow at BFPG
- Brisbane Functional Programming Group
- Tony Morris' Programming Blog

These are some of the resources I used when putting this presentation together and the link to this presentation. If you print out the presentation you will get all of my handout notes, which is basically a narrative of everything I've said!

I've already mentioned the Brisbane Functional Programming Group, but if you want to learn more about functional programming, your best bet is to join that group and start participating. It can be challenging at times because there are some very smart and knowledgable people there, but they really want to help people like you and I come to terms with functional programming and are more than willing to help answer questions.

Finally there is Tony Morris' programming blog which is always an entertaining read. Tony has his point of view and isn't afraid to wield it, but I wouldn't have been able to put this presentation together if it wasn't for the generosity of Tony taking the time to teach people like myself a bit more about this stuff. so thank you Tony!