Functional Programming For The Rest of Us

Jun 19, 2006

Introduction

Programmers are procrastinators. Get in, get some coffee, check the mailbox, read the RSS feeds, read the news, check out latest articles on techie websites, browse through political discussions on the designated sections of the programming forums. Rinse and repeat to make sure nothing is missed. Go to lunch. Come back, stare at the IDE for a few minutes. Check the mailbox. Get some coffee. Before you know it, the day is over.

The only thing, every once in a while challenging articles actually do pop up. If you're looking at the right places you'll find at least one of these every couple of days. These articles are hard to get through and take some time, so they start piling up. Before you know it, you have a list of links and a folder full of PDF files and you wish you had a year in a small hut in the middle of the forest with nobody around for miles so you could catch up. Would be nice if someone came in every morning while you're taking a walk down the river to bring some food and take out the garbage.

I don't know about your list, but a large chunk of the articles in mine are about functional programming. These generally are the hardest to get through. Written in a dry academic language, even the "ten year Wall Street industry veterans" don't understand what functional programming (also referred to as *FP*) articles are all about. If you ask a project manager in Citi Group or in Deutsche Bank[1] why they chose to use JMS instead of Erlang they'll say they can't use academic languages for industrial strength applications. The problem is, some of the most complex systems with the most rigid requirements are written using functional programming elements. Something doesn't add up.

It's true that FP articles and papers are hard to understand, but they don't have to be. The reasons for the knowledge gap are purely historical. There is nothing inherently hard about FP concepts. Consider this article "an accessible guide to FP", a bridge from our imperative minds into the world of FP. Grab a coffee and keep on reading. With any luck your coworkers will start making fun of you for your FP comments in no time.

So what is FP? How did it come about? Is it edible? If it's as useful as its advocates claim, why isn't it being used more often in the industry? Why is it that only people with PhDs tend to use it? Most importantly, why is it so damn hard to learn? What is all this closure, continuation, currying, lazy evaluation and no side effects business? How can it be used in projects that don't involve a university? Why does it seem to be so different from everything good, and holy, and dear to our imperative hearts? We'll clear this up very soon. Let's start with explaining the reasons for the huge gap between the real world and academic articles. The answer is as easy as taking a walk in the park.

A Walk In The Park

Fire up the time machine. Our walk in the park took place more than two thousand years ago, on a beautiful sunny day of a long forgotten spring in 380 B.C. Outside the city walls of Athens, under the pleasant shade of olive trees Plato was walking towards the Academy with a beautiful slave boy. The weather was lovely, the dinner was filling, and the conversation turned to philosophy.

"Look at these two students", said Plato carefully picking words to make the question educational. "Who do you think is taller?" The slave boy looked towards the basin of water where two men were standing. "They're about the same height", he said. "What do you mean 'about the same'?", asked Plato. "Well, they look the same from here but I'm sure if I were to get closer I'd see that there is some difference."

Plato smiled. He was leading the boy in the right direction. "So you would say that there is nothing perfectly equal in our world?" After some thinking the boy replied: "I don't think so. Everything is at least a little different, even if we can't see it." The point hit home! "Then if nothing is perfectly equal in this world, how do you think you understand the concept of 'perfect' equality?" The slave boy looked puzzled. "I don't know", he replied.

So was born the first attempt to understand the nature of mathematics. Plato suggested that everything in our world is just an approximation of perfection. He also realized that we understand the concept of perfection even though we never encountered it. He came to conclusion that perfect mathematical forms must live in another world and that we somehow know about them by having a connection to that "alternative" universe. It's fairly clear that there is no perfect circle that we can observe. But we also understand what a perfect circle is and can describe it via equations. What is mathematics, then? Why is the universe described with mathematical laws? Can all of the phenomena of our universe be described by mathematics?[2]

Philosophy of mathematics is a very complex subject. Like most philosophical disciplines it is far more adept at posing questions rather than providing answers. Much of the consensus revolves around the fact that mathematics is really a puzzle: we set up a set of basic non-conflicting principles and a set of rules on how to operate with these principles. We can then stack these rules together to come up with more complex rules. Mathematicians call this method a "formal system" or a "calculus". We can effectively write a formal system for Tetris if we wanted to. In fact, a working implementation of Tetris *is* a formal system, just specified using an unusual representation.

A civilization of furry creatures on Alpha Centauri would not be able to read our formalisms of Tetris and circles because their only sensory input might be an organ that senses smells. They likely will never find out about the Tetris formalism, but they very well might have a formalism for circles. We probably wouldn't be able to read it because our sense of smell isn't that sophisticated, but once you get past the *representation* of the formalism (via various sensory instruments and standard code breaking techniques to understand the *language*), the concepts underneath are understandable to any intelligent civilization.

Interestingly if no intelligent civilization ever existed in the universe the formalisms for Tetris and circles would still hold water, it's just that nobody would be around to find out about them. If an intelligent civilization popped up, it would likely discover some formalisms that help describe the laws of our universe. They also would be very unlikely to ever find out about Tetris because there is nothing in the universe that resembles it. Tetris is one of countless examples of a formal system, a puzzle, that has nothing to do with the real world. We can't even be sure that natural numbers have full resemblance to the real world, after all one can easily think of a number so big that it cannot describe anything in our universe since it might actually turn out to be finite.

A Bit of History[3]

Let's shift gears in our time machine. This time we'll travel a lot closer, to the 1930s. The Great Depression was ravaging the New and the Old worlds. Almost every family from every social class was affected by the tremendous economic downturn. Very few sanctuaries remained where people were safe from the perils of poverty. Few people were fortunate enough to be in these sanctuaries, but they did exist. Our interest lies in mathematicians in Princeton University.

The new offices constructed in gothic style gave Princeton an aura of a safe haven. Logicians from all over the world were invited to Princeton to build out a new department. While most of America couldn't find a piece of bread for dinner, high ceilings, walls covered with elaborately carved wood, daily discussions by a cup of tea, and walks in the forest were some of the conditions in Princeton.

One mathematician living in such lavish lifestyle was a young man named Alonzo Church. Alonzo received a B.S. degree from Princeton and was persuaded to stay for graduate school. Alonzo felt the architecture was fancier than necessary. He rarely showed up to discuss mathematics with a cup of tea and he didn't enjoy the walks in the woods. Alonzo was a loner:

he was most productive when working on his own. Nevertheless Alonzo had regular contacts with other Princeton inhabitants. Among them were Alan Turing, John von Neumann, and Kurt Gödel.

The four men were interested in formal systems. They didn't pay much heed to the physical world, they were interested in dealing with abstract mathematical puzzles instead. Their puzzles had something in common: the men were working on answering questions about *computation*. If we had machines that had infinite computational power, what problems would we be able to solve? Could we solve them automatically? Could some problems remain unsolved and why? Would various machines with different designs be equal in power?

In cooperation with other men Alonzo Church developed a formal system called *lambda calculus*. The system was essentially a programming language for one of these imaginary machines. It was based on functions that took other functions as parameters and returned functions as results. The function was identified by a Greek letter lambda, hence the system's name[4]. Using this formalism Alonzo was able to reason about many of the above questions and provide conclusive answers.

Independently of Alonzo Church, Alan Turing was performing similar work. He developed a different formalism (now referred to as the Turing machine), and used it to independently come to similar conclusions as Alonzo. Later it was shown that Turing machines and lambda calculus were equivalent in power.

This is where the story would stop, I'd wrap up the article, and you'd navigate to another page, if not for the beginning of World War II. The world was in flames. The U.S. Army and Navy used artillery more often than ever. In attempts to improve accuracy the Army employed a large group of mathematicians to continuously calculate differential equations required for solving ballistic firing tables. It was becoming obvious that the task was too great for being solved manually and various equipment was developed in order to overcome this problem. The first machine to solve ballistic tables was a Mark I built by IBM - it weighed five tons, had 750,000 parts and could do three operations per second.

The race, of course, wasn't over. In 1949 an Electronic Discrete Variable Automatic Computer (*EDVAC*) was unveiled and had tremendous success. It was a first example of von Neumann's architecture and was effectively a real world implementation of a Turing machine. For the time being Alonzo Church was out of luck.

In late 1950s an MIT professor John McCarthy (also a Princeton graduate) developed interest in Alonzo Church's work. In 1958 he unveiled a List Processing language (Lisp). Lisp was an implementation of Alonzo's lambda calculus that worked on von Neumann computers! Many computer scientists recognized the expressive power of Lisp. In 1973 a group of programmers at MIT's Artificial Intelligence Lab developed hardware they called a Lisp machine - effectively a native hardware implementation of Alonzo's lambda calculus!

Functional Programming

Functional programming is a practical implementation of Alonzo Church's ideas. Not all lambda calculus ideas transform to practice because lambda calculus was not designed to work under physical limitations. Therefore, like object oriented programming, functional programming is a set of ideas, not a set of strict guidelines. There are many functional programming languages, and most of them do many things very differently. In this article I will explain the most widely used ideas from functional languages using examples written in Java (yes, you could write functional programs in Java if you felt particularly masochistic). In the next couple of sections we'll take Java as is, and will make modifications to it to transform it into a useable functional language. Let's begin our quest.

Lambda calculus was designed to investigate problems related to *calculation.* Functional programming, therefore, primarily deals with calculation, and, surprisingly, uses functions to do so. A function is a very basic unit in functional programming. Functions are used for almost everything, even the simplest of calculations. Even variables are replaced with functions. In functional programming variables are simply aliases for expressions (so we don't have to type everything on one line). They cannot be modified. All variables can only be assigned to once. In Java terms this means that every single variable is declared as *final* (or *const* if we're dealing with C++). There are no non-final variables in FP.

```
final int i = 5;
final int j = i + 3;
```

Since every variable in FP is final two fairly interesting statements can be made. It does not make sense to always write the keyword *final* and it does not make sense to call variables, well... variables. We will now make two modifications to Java: every variable declared in our functional Java will be final by default, and we will refer to variables as *symbols*.

By now you are probably wondering how you could possibly write anything reasonably complicated in our newly created language. If every symbol is non-mutable we cannot change the state of anything! This isn't strictly true. When Alonzo was working on lambda calculus he wasn't interested in maintaining state over periods of time in order to modify it later. He was interested in performing operations on data (also commonly referred to as "calculating stuff"). However, it was proved that lambda calculus is equivalent to a Turing machine. It can do all the same things an imperative programming language can. How, then, can we achieve the same results?

It turns out that functional programs can keep state, except they don't use variables to do it. They use functions instead. The state is kept in function parameters, on the stack. If you want to keep state for a while and every now and then modify it, you write a recursive function. As an example, let's write a function that reverses a Java string. Remember, every variable we declare is final by default[5].

```
String reverse(String arg) {
    if(arg.length == 0) {
        return arg;
    }
    else {
        return reverse(arg.substring(1, arg.length)) + arg.substring(0, 1)
    }
}
```

This function is slow because it repeatedly calls itself[6]. It's a memory hog because it repeatedly allocates objects. But it's functional in style. You may be interested why someone would want to program in this manner. Well, I was just about to tell you.

Benefits of FP

You're probably thinking that there's no way I can rationalize the monstrosity of a function above. When I was learning functional programming I was thinking that too. I was wrong. There are very good arguments for using this style. Some of them are subjective. For example, people claim that functional programs are easier to understand. I will leave out these arguments because every kid on the block knows that ease of understanding is in the eye of the beholder. Fortunately for me, there are plenty of objective arguments.

Unit Testing

Since every symbol in FP is final, no function can ever cause side effects. You can never modify things in place, nor can one function modify a value outside of its scope for another function to use (like a class member or a global variable). That means that the only effect of evaluating a function is its return value and the only thing that affects the return value of a function is its arguments.

This is a unit tester's wet dream. You can test every function in your program only worrying about its arguments. You don't have to worry about calling functions in the right order, or setting up external state properly. All you need to do is pass arguments that represent edge cases. If every function in your program passes unit tests you can be a lot more confident about quality of your software than if you were using an imperative language. In Java or C++ checking a return value of a function is not sufficient - it may modify external state that we would need to verify. Not so in a functional language.

Debugging

If a functional program doesn't behave the way you expect it to, debugging it is a breeze. You will always be able to reproduce your problem because a bug in a functional program doesn't depend on unrelated code paths that were executed before it. In an imperative program a bug resurfaces only some of the time. Because functions depend on external state produced by side effects from other functions you may have to go through a series of steps in no way related to the bug. In a functional program this isn't the case - if a return value of a function is wrong, it is *always* wrong, regardless of what code you execute before running the function.

Once you reproduce the problem, getting to the bottom of it is trivial. It is almost pleasant. You break the execution of your program and examine the stack. Every argument in every function call in the stack is available for your inspection, just like in an imperative program. Except in an imperative program that's not enough because functions depend on member variables, global variables, and the state of other classes (which in turn depend on these very same things). A function in a functional program depends *only* on its arguments, and that information is right before your eyes! Furthermore, in an imperative program examining a return value of a function will not give you a good idea of whether the function behaves properly. You need to hunt down dozens of objects outside its scope to verify that it performed correct actions. In a functional program all you have to do is look at the return value!

Walking through the stack you look at arguments passed to functions and their return values. The minute a return value doesn't make sense you step into the offending function and walk through it. You repeat this recursively until the process leads you to the source of the bug!

Concurrency

A functional program is ready for concurrency without any further modifications. You never have to worry about deadlocks and race conditions because you don't need to use locks! No piece of data in a functional program is modified twice by the same thread, let alone by two different threads. That means you can easily add threads without ever giving conventional problems that plague concurrency applications a second thought!

If this is the case, why doesn't anybody use functional programs for highly concurrent applications? Well, it turns out that they do. Ericsson designed a functional language called Erlang for use in its highly tolerant and scalable telecommunication switches. Many others recognized the benefits provided by Erlang and started using it. We're talking about telecommunication and traffic control systems that are far more scalable and reliable than typical systems designed on Wall Street. Actually, Erlang systems are not scalable and reliable. Java systems are. Erlang systems are simply rock solid.

The concurrency story doesn't stop here. If your application is inherently single threaded the compiler can still optimize functional programs to run on multiple CPUs. Take a look at the following code fragment:

```
String s1 = somewhatLongOperation1();
String s2 = somewhatLongOperation2();
String s3 = concatenate(s1, s2);
```

In a functional language the compiler could analyze the code, classify the functions that create strings *s1* and *s2* as potentially time consuming operations, and run them concurrently. This is impossible to do in an imperative language because each function may modify state outside of its scope and the function following it may depend on it. In functional languages automatic analysis of functions and finding good candidates for concurrent execution is as trivial as automatic inlining! In this sense functional style programs are "future proof" (as much as I hate buzzwords, I'll indulge this time). Hardware manufacturers can no longer make CPUs run any faster. Instead they increase the number of cores and attribute quadruple speed increases to concurrency. Of course they conveniently forget to mention that we get our money's worth only on software that deals with parallelizable problems. This is a very small fraction of imperative software but 100% of functional software because functional programs are all parallelizable out of the box.

Hot Code Deployment

In the old days of Windows in order to install updates it was necessary to restart the computer. Many times. After installing a newer version of a media players. With Windows XP the situation has improved significantly, yet it still isn't ideal (I ran Windows Update at work today and now an annoying system tray icon won't go away until I restart). Unix systems have had a better model for a while. In order to install an update you only need to stop relevant components, not the whole OS. While it is a better situation, for a large class of server applications it still isn't acceptable. Telecommunication systems need to be up 100% of the time because if dialing emergency is not available due to upgrades, lives may be lost. There is no reason Wall Street firms need to bring down their systems to install software updates over the weekend.

An ideal situation is updating relevant parts of the code without stopping any part of the system at all. In an imperative world this isn't possible. Consider unloading a Java class at runtime and reloading a new definition. If we were to do that every instance of a class would become unusable because the state it holds would be lost. We would need to resort to writing tricky version control code. We'd need to serialize all running instances of the class, destroy them, create instances of the new class, try to load serialized data into them hoping the loading code properly migrates the data to work with the new instance. On top of that, every time we change something we'd have to write our migration code manually. And our migration code would have to take special care not to break relationships between objects. Nice in theory, but would never work well in practice.

In a functional program all state is stored on the stack in the arguments passed to functions. This makes hot deployment significantly easier! In fact, all we'd really have to do is run a diff between the code in production and the new version, and deploy the new code. The rest could be done by language tools automatically! If you think this is science fiction, think again. Erlang engineers have been upgrading live systems without stopping them for years.

Machine Assisted Proofs and Optimizations

An interesting property of functional languages is that they can be reasoned about mathematically. Since a functional language is simply an implementation of a formal system, all mathematical operations that could be done on paper still apply to the programs written in that language. The compiler could, for example, convert pieces of code into equivalent but more efficient pieces with a mathematical proof that two pieces of code are equivalent[7]. Relational databases have been performing these optimizations for years. There is no reason the same techniques can't apply to regular software.

Additionally, you can use these techniques to prove that parts of your program are correct. It is even possible to create tools that analyze code and generate edge cases for unit tests automatically! This functionality is invaluable for rock solid systems. If you are designing pace makers and air traffic control systems such tools are almost always a requirement. If you are writing an application outside of truly mission critical industries, these tools can give you a tremendous edge over your competitors.

Higher Order Functions

I remember learning about the benefits I outlined above and thinking "that's all very nice but it's useless if I have to program in a crippled language where everything is final." This was a misconception. Making all variables final *is* crippled in a context of an imperative language like Java but it isn't in a context of functional languages. Functional languages offer a different kind of abstraction tools that make you forget you've ever *liked* modifying variables. One such tool is capability to work with *higher order functions*.

A function in such languages is different from a function in Java or C. It is a superset - it can do all the things a Java function can do, and more. We create a function in the same manner we do in C:

```
int add(int i, int j) {
    return i + j;
}
```

This means something different from equivalent C code. Let's extend our Java compiler to support this notation. When we type something like this our compiler will convert it to the following Java code (don't forget, everything is final):

```
class add_function_t {
    int add(int i, int j) {
        return i + j;
    }
}

add_function_t add = new add_function_t();
```

The symbol *add* isn't really a function. It is a small class with one function as its member. We can now pass *add* around in our code as an argument to other functions. We can assign it to another symbol. We can create instances of *add_function_t* at runtime and they will be garbage collected when we no longer need them. This makes functions *first class objects* no different from integers or strings. Functions that operate on other functions (accept them as arguments) are called *higher order functions*. Don't let this term intimidate you, it's no different from Java classes that operate on each other (we can pass class instances to other classes). We can call them "higher order classes" but nobody cares to because there is no strong academic community behind Java.

How, and when, do you use higher order functions? Well, I'm glad you asked. You write your program as a big monolithic blob of code without worrying about class hierarchies. When you see that a particular piece of code is repeated, you break it out into a function (fortunately they still teach this in schools). If you see that a piece of logic within your function needs to behave differently in different situations, you break it out into a higher order function. Confused? Here's a real life example from my work.

Suppose we have a piece of Java code that receives a message, transforms it in various ways, and forwards it to another server.

```
class MessageHandler {
    void handleMessage(Message msg) {
        // ...
        msg.setClientCode("ABCD_123");
        // ...

        sendMessage(msg);
    }

    // ...
}
```

Now imagine that our system has changed and we now route messages to two servers instead of one. Everything is handled in exactly the same way except the client code - the second server wants it in a different format. How do we handle this situation? We could check where the message is headed and format the client code differently, like this:

```
class MessageHandler {
    void handleMessage(Message msg) {
        // ...
        if(msg.getDestination().equals("server1") {
            msg.setClientCode("ABCD_123");
        } else {
            msg.setClientCode("123_ABC");
        }
        // ...

        sendMessage(msg);
    }

    // ...
}
```

This approach, however, isn't scalable. If more servers are added our function will grow linearly and we'll have a nightmare updating it. An object oriented approach is to make *MessageHandler* a base class and specialize the client code operation in derived classes:

```
abstract class MessageHandler {
    void handleMessage(Message msg) {
        // ...
        msg.setClientCode(getClientCode());
        // ...

        sendMessage(msg);
    }

    abstract String getClientCode();

    // ...
}

class MessageHandlerOne extends MessageHandler {
    String getClientCode() {
        return "ABCD_123";
    }
}

class MessageHandlerTwo extends MessageHandler {
    String getClientCode() {
        return "123_ABCD";
    }
}
```

We can now instantiate an appropriate class for each server. Adding servers becomes much more maintainable. That's a lot of code for such a simple modification though. We have to create two new types just to support different client codes! Now let's do the same thing in our language that supports higher order functions:

```
class MessageHandler {
    void handleMessage(Message msg, Function getClientCode) {
        // ...
        Message msg1 = msg.setClientCode(getClientCode());
        // ...

        sendMessage(msg1);
    }

    // ...
}

String getClientCodeOne() {
    return "ABCD_123";
}

String getClientCodeTwo() {
    return "123_ABCD";
}

MessageHandler handler = new MessageHandler();
handler.handleMessage(someMsg, getClientCodeOne);
```

We've created no new types and no class hierarchy. We simply pass appropriate functions as a parameter. We've achieved the same thing as the object oriented counterpart with a number of advantages. We don't restrict ourselves to class hierarchies: we can pass new functions at runtime and change them at any time with a much higher degree of granularity with less code. Effectively the compiler has written object oriented "glue" code for us! In addition we get all the other benefits of FP. Of course the abstractions provided by functional languages don't stop here. Higher order functions are just the beginning.

Currying

Most people I've met have read the Design Patterns book by the Gang of Four. Any self respecting programmer will tell you that the book is language agnostic and the patterns apply to software engineering in general, regardless of which language you use. This is a noble claim. Unfortunately it is far removed from the truth.

Functional languages are extremely expressive. In a functional language one does not need design patterns because the language is likely so high level, you end up programming in concepts that eliminate design patterns all together. Once such pattern is an Adapter pattern (how is it different from Facade again? Sounds like somebody needed to fill more pages to satisfy their contract). It is eliminated once a language supports a technique called *currying*.

Adapter pattern is best known when applied to the "default" abstraction unit in Java - a class. In functional languages the pattern is applied to functions. The pattern takes an interface and transforms it to another interface someone else expects. Here's an example of an adapter pattern:

```
int pow(int i, int j);
int square(int i)
{
    return pow(i, 2);
}
```

The code above adapts an interface of a function that raises an integer to an integer power to an interface of a function that squares an integer. In academic circles this trivial technique is called currying (after a logician Haskell Curry who performed mathematical acrobatics necessary to formalize it). Because in FP functions (as opposed to classes) are passed around as arguments, currying is used very often to adapt functions to an interface that someone else expects. Since the interface to functions is its arguments, currying is used to reduce the number of arguments (like in the example above).

Functional languages come with this technique built in. You don't need to manually create a function that wraps the original, functional languages will do that for you. As usual, let's extend our language to support this technique.

```
square = int pow(int i, 2);
```

```
            <p>This will automatically create a function <em>square</em> f
            the second argument set to <em>2</em>. This will get compiled
```

```
class square_function_t {
    int square(int i) {
        return pow(i, 2);
    }
}
square_function_t square = new square_function_t();
```

As you can see, we've simply created a wrapper for the original function. In FP currying is just that - a shortcut to quickly and easily create wrappers. You concentrate on your task, and the compiler writes the appropriate code for you! When do you use currying? This should be easy. Any time you'd like to use an adapter pattern (a wrapper).

Lazy Evaluation

Lazy (or delayed) evaluation is an interesting technique that becomes possible once we adopt a functional philosophy. We've already seen the following piece of code when we were talking about concurrency:

```
String s1 = somewhatLongOperation1();
String s2 = somewhatLongOperation2();
String s3 = concatenate(s1, s2);
```

In an imperative language the order of evaluation would be clear. Because each function may affect or depend on an external state it would be necessary to execute them in order: first *somewhatLongOperation1*, then *somewhatLongOperation2*, followed by *concatenate*. Not so in functional languages.

As we saw earlier *somewhatLongOperation1* and *somewhatLongOperation2* can be executed concurrently because we're guaranteed no function affects or depends on global state. But what if we don't want to run the two concurrently, do we need to run them in order? The answer is no. We only need to run these operations when another function depends on *s1* and *s2*. We don't even have to run them before *concatenate* is called - we can delay their evaluation until they're required within *concatenate*. If we replace *concatenate* with a function that has a conditional and uses only one of its two parameters we may never evaluate one of the parameters at all! *Haskell* is an example of a delayed evaluation language. In *Haskell* you are not guaranteed that anything will be executed in order (or at all) because *Haskell* only executes code when it's required.

Lazy evaluation has numerous advantages as well as disadvantages. We will discuss the advantages here and will explain how to counter the disadvantages in the next section.

Optimization

Lazy evaluation provides a tremendous potential for optimizations. A lazy compiler thinks of functional code exactly as mathematicians think of an algebra expression - it can cancel things out and completely prevent execution, rearrange pieces of code for higher efficiency, even arrange code in a way that reduces errors, all guaranteeing optimizations won't break the code. This is the biggest benefit of representing programs strictly using formal primitives - code adheres to mathematical laws and can be reasoned about mathematically.

Abstracting Control Structures

Lazy evaluation provides a higher order of abstraction that allows implementing things in a way that would otherwise be impossible. For example consider implementing the following control structure:

```
unless(stock.isEuropean()) {
    sendToSEC(stock);
}
```

We want *sendToSEC* executed unless the stock is European. How can we implement *unless*? Without lazy evaluation we'd need some form of a macro system, but in a language like Haskell that's unnecessary. We can implement *unless* as a function!

```
void unless(boolean condition, List code) {
    if(!condition)
        code;
}
```

Note that *code* is never evaluated if the condition is true. We cannot reproduce this behavior in a strict language because the arguments would be evaluated before *unless* is entered.

Infinite Data Structures

Lazy languages allow for definition of infinite data structures, something that's much more complicated in a strict language. For example, consider a list with Fibonacci numbers. We clearly can't compute and infinite list in a reasonable amount of time or store it in memory. In strict languages like Java we simply define a Fibonacci function that returns a particular member from the sequence. In a language like Haskell we can abstract it further and simply define an infinite list of Fibonacci numbers. Because the language is lazy, only the necessary parts of the list that are actually used by the program are ever evaluated. This allows for abstracting a lot of problems and looking at them from a higher level (for example, we can use list processing functions on an infinite list).

Disadvantages

Of course there ain't no such thing as a free lunch(tm). Lazy evaluation comes with a number of disadvantages. Mainly that it is, well, lazy. Many real world problems require strict evaluation. For example consider the following:

```
System.out.println("Please enter your name: ");
System.in.readLine();
```

In a lazy language you have no guarantee that the first line will be executed before the second! This means we can't do IO, can't use native functions in any meaningful way (because they need to be called in order since they depend on side effects), and can't interact with the outside world! If we were to introduce primitives that allow ordered code execution we'd lose the benefits of reasoning about our code mathematically (which would take all of the benefits of functional programming with it). Fortunately not all is lost. Mathematicians got to work and developed a number of tricks to ensure code gets executed in particular order in a functional setting. We get the best of both worlds! These techniques include continuations, monads, and uniqueness typing. In this article we'll only deal with continuations. We'll leave monads and uniqueness typing for another time. Interestingly, continuations are useful for many things other than enforcing a particular order of evaluation. We'll talk about that as well.

Continuations

Continuations to programming are what Da Vinci Code is to human history: an amazing revelation of the greatest cover-up known to man. Well, may be not, but they're certainly revealing of deceit in the same sense as square roots of negative numbers.

When we learned about functions we only learned half truths based on a faulty assumption that functions must return their value to the original caller. In this sense continuations are a generalization of functions. A function must not necessarily return to its caller and may return to any part of the program. A "continuation" is a parameter we may choose to pass to our function that specifies where the function should return. The description may be more complicated than it sounds. Take a look at the following code:

```
int i = add(5, 10);
int j = square(i);
```

The function *add* returns *15* to be assigned to *i*, the place where *add* was originally called. After that the value of *i* is used to call *square*. Note that a lazy compiler can't rearrange these lines of code because the second line depends on successful evaluation of the first. We can rewrite this code block using *Continuation Passing Style* or *CPS*, where the function *add* doesn't return to the original caller but instead returns its result to *square*.

```
int j = add(5, 10, square);
```

In this case *add* gets another parameter - a function that *add* must call with its result upon completion. In this case *square* is a continuation of *add*. In both cases *j* will equal *225*.

Here lays the first trick to force a lazy language to evaluate two expressions in order. Consider the following (familiar) IO code:

```
System.out.println("Please enter your name: ");
System.in.readLine();
```

The two lines don't depend on each other and the compiler is free to rearrange them as it wishes. However, if we rewrite this code in CPS, there will be a dependency and the compiler will be forced to evaluate the two lines in order!

```
System.out.println("Please enter your name: ", System.in.readLine);
```

In this case *println* needs to call *readLine* with its result and return the result of *readLine*. This allows us to ensure that the two lines are executed in order *and* that *readLine* is evaluated at all (because the whole computation expects the last value as a result). In case of Java *println* returns *void* but if it were to return an abstract value (that *readLine* would accept), we'd solve our problem! Of course chaining function calls like that will quickly become unreadable, but it isn't necessary. We could add syntactic sugar to the language that will allow us to simply type expressions in order, and the compiler would chain the calls for us automatically. We can now evaluate expressions in any order we wish without losing any of the benefits of FP (including the ability to reason about our programs mathematically)! If this is still confusing, remember that a function is just an instance of a class with one member. Rewrite above two lines so that *println* and *readLine* are instances of classes and everything will become clear.

I would now wrap up this section, except that we've only scratched the surface of continuations and their usefulness. We can write entire programs in CPS, where every function takes an extra continuation argument and passes the result to it. We can also convert any program to CPS simply by treating functions as special cases of continuations (functions that always return to their caller). This conversion is trivial to do automatically (in fact, many compilers do just that).

Once we convert a program to CPS it becomes clear that every instruction has some continuation, a function it will call with the result, which in a regular program would be a place it must return to. Let's pick any instruction from above code, say *add(5, 10)*. In a program written in CPS style it's clear what *add*'s continuation is - it's a function that *add* calls once it's done. But what is it in a non-CPS program? We could, of course, convert the program to CPS, but do we have to?

It turns out that we don't. Look carefully at our CPS conversion. If you try to write a compiler for it and think about it long enough you'll realize that the CPS version needs no stack! No function ever "returns" in the traditional sense, it just calls another function with the result instead. We don't need to push function arguments on the stack with every call and then pop them back, we can simply store them in some block of memory and use a jump instruction instead. We'll never need the original arguments - they'll never be used again since no function ever returns!

So, programs written in CPS style have no stack but have an extra argument with a function to call. Programs not written in CPS style have no argument with a function to call, but have the stack instead. What does the stack contain? Simply the arguments, and a pointer to memory where the function should return. Do you see a light bulb? The stack simply contains continuation information! The pointer to the return instruction in the stack is essentially the same thing as the function to call in CPS programs! If you wanted to find out what continuation for *add(5, 10)* is, you'd simply have to examine the stack at the point of its execution!

So that was easy. A continuation and a pointer to the return instruction in the stack are really the same thing, only a continuation is passed explicitly, so that it doesn't need to be the same place where the function was called from. If you remember that a continuation is a function, and a function in our language is compiled to an instance of a class, you'll realize that a pointer to the return instruction in the stack and the continuation argument are *really* the same thing, since our function (just like an instance of a class) is simply a pointer. This means that at any given point in time in your program you can ask for a *current continuation* (which is simply the information on the stack).

Ok, so we know what a current continuation is. What does it mean? When we get a current continuation and store it somewhere, we end up storing the current state of our program - freezing it in time. This is similar to an OS putting itself into hibernation. A continuation object contains the information necessary to restart the program from the point where the continuation object was acquired. An operating system does this to your program all the time when it context switches between the threads. The only difference is that it keeps all the control. If you ask for a continuation object (in Scheme this is done by calling *call-with-current-continuation* function) you'll get an object that contains the current continuation - the stack (or in a CPS case the function to call next). You can store this object in a variable (or alternatively, on disk). When you choose to "restart" your program with this continuation object you will "transform" to the state of the program when you grabbed the continuation object. It's the same thing as switching back to a suspended thread or waking up an OS from hibernation, except you can do it again and again. When an OS wakes up, the hibernation information is destroyed. If it wasn't, you'd be able to wake up from the same point over and over again, almost like going back in time. You have that control with continuations!

In what situations are continuations useful? Usually when you're trying to simulate state in an application of inherently stateless nature to ease your life. A great application of continuations are web applications. Microsoft's ASP.NET goes to tremendous lengths to try and simulate state so that you can write your application with less hassle. If C# supported continuations half of ASP.NET's complexity would disappear - you'd simply store a continuation and restart it when a user makes the web request again. To a programmer of the web application there would be no interruption - the program would simply start from the next line! Continuations are an incredibly useful abstraction for some problems. Considering that many of the traditional fat clients are moving to the web, continuations will become more and more important in the future.

```
        <h2><a id="part_10">Pattern Matching</a></h2>
```

Pattern matching is not a new or innovative feature. In fact, it has little to do with functional programming. The only reason why it's usually attributed to FP is that functional languages have had pattern matching for some time, while modern imperative languages still don't.

Let's dive into pattern matching with an example. Here's a Fibonacci function in Java:

```
int fib(int n) {
    if(n == 0) return 1;
    if(n == 1) return 1;

    return fib(n - 2) + fib(n - 1);
}
```

And here's an example of a Fibonacci function in our Java-derived language that supports pattern matching:

```
int fib(0) {
    return 1;
}
int fib(1) {
    return 1;
}
int fib(int n) {
    return fib(n - 2) + fib(n - 1);
}
```

What's the difference? The compiler implements branching for us.

What's the big deal? There isn't any. Someone noticed that a large number of functions contain very complicated switch statements (this is particularly true about functional programs) and decided that it's a good idea to abstract that away. We split the function definition into multiple ones, and put patterns in place of some arguments (sort of like overloading). When the function is called, the compiler compares the arguments with the definitions at runtime, and picks the correct one. This is usually done by picking the most specific definition available. For example, *int fib(int n)* can be called with *n* equal to *1*, but it isn't because *int fib(1)* is more specific.

Pattern matching is usually more complex than our example reveals. For example, an advanced pattern matching system will allow us to do the following:

```
int f(int n < 10) { ... }
int f(int n) { ... }
```

When is pattern matching useful? In a surprisingly large number of cases! Every time you have a complex structure of nested *if*s, pattern matching can do a better job with less code on your part. A good function that comes to mind is a standard *WndProc* function that all Win32 applications must provide (even though it's often abstracted away). Usually a pattern matching system can examine collections as well as simple values. For example, if you pass an array to your function you could pick out all arrays in which the first element is equal to *1* and the third element is greater than *3*.

Another benefit of pattern matching is that if you need to add or modify conditions, you don't have to go into one huge function. You simply add (or modify) appropriate definitions. This eliminates the need for a whole range of design patterns from the GoF book. The more complex your conditions are, the more pattern matching will help you. Once you're used to it, you start wondering how you ever got through your day without it.

Closures

So far we've discussed features in the context of "pure" functional languages - languages that are implementations of lambda calculus and don't include features that conflict with Church's formalism. However, many of the features of functional languages are useful outside of lambda calculus framework. While an implementation of an axiomatic system is useful because it allows thinking about programs in terms of mathematical expressions, it may or may not always be practical. Many languages choose to incorporate functional elements without strictly adhering to functional doctrine. Many such languages (like Common Lisp) don't require variables to be final - you can modify things in place. They also don't require functions to depend only on their arguments - functions are allowed to access state outside of their boundaries. But they do include functional features - like higher order functions. Passing functions around in impure languages is a little bit different than doing it in the confines of lambda calculus and requires support for an interesting feature often referred to as lexical closure. Let's take a look at some sample code. Remember, in this case variables aren't final and functions can refer to variables outside of their scope:

```
Function makePowerFn(int power) {
   int powerFn(int base) {
       return pow(base, power);
   }

   return powerFn;
}

Function square = makePowerFn(2);
square(3); // returns 9
```

The function *make-power-fn* returns a function that takes a single argument and raises it to a certain power. What happens when we try to evaluate *square(3)*? The variable *power* isn't anywhere in scope of *powerFn* because *makePowerFn* has returned and its stack is long gone. How can *square* work, then? The language must, somehow, store the value of *power* somewhere for *square* to work. What if we create another function, *cube*, that raises something to the third power? The runtime must now store two copies of power, one for each function we generated using *make-power-fn*. The phenomenon of storing these values is called a *closure*. Closures don't only store arguments of a host function. For example, a closure can look like this:

```
Function makeIncrementer() {
   int n = 0;

   int increment() {
       return ++n;
   }
}

Function inc1 = makeIncrementer();
Function inc2 = makeIncrementer();

inc1(); // returns 1;
inc1(); // returns 2;
inc1(); // returns 3;
inc2(); // returns 1;
inc2(); // returns 2;
inc2(); // returns 3;
```

The runtime manages to store *n*, so incrementers can access it. Furthermore, it stores various copies, one for each incrementer, even though they're supposed to disappear when *makeIncrementer* returns. What does this code compile to? How do closures work behind the scenes? Fortunately, we have a back stage pass.

A little common sense goes a long way. The first observation is that local variables are no longer limited to simple scope rules and have an undefined lifetime. The obvious conclusion is that they're no longer stored on the stack - they must be stored on the heap instead[8]. A closure, then, is implemented just like a function we discussed earlier, except that it has an additional reference to the surrounding variables:

```
class some_function_t {
   SymbolTable parentScope;

   // ...
}
```

When a closure references a variable that's not in its local scope, it consults this reference for a parent scope. That's it! Closures bring functional and OO worlds closer together. Every time you create a class that holds some state and pass it to somewhere else, think of closures. A closure is just an object that creates "member variables" on the fly by grabbing them from the scope, so you don't have to!

What's next?

This article only scratches the surface of functional programming. Sometimes a small scratch can progress to something bigger and in our case it's a good thing. In the future I plan to write about category theory, monads, functional data structures, type systems in functional languages, functional concurrency, functional databases and much more. If I get to write (and in the process learn) about half of these topics my life will be complete. In the meantime, Google is our friend.

Comments?

If you have any questions, comments, or suggestions, please drop a note at coffeemug@gmail.com. I'll be glad to hear your feedback.

---

[1]*When I was looking for a job in the fall of 2005 I often did ask this question. It's quite amusing how many blank stares I got. You would think that at about $300,000 a piece these people would at least have a good understanding of most tools available to them.*

[2]*This appears to be a controversial question. Physicists and mathematicians are forced to acknowledge that it isn't at all clear whether everything in the universe obeys the laws that can be described by mathematics.*

[3] *I've always hated history lessons that offer a dry chronology of dates, names, and events. To me history is about the lives of people who changed the world. It is about their private reasons behind their actions, and the mechanisms by which they affected millions of souls. For this reason this history section is hopelessly incomplete. Only very relevant people and events are discussed.*

[4] *When I was learning about functional programming I was very annoyed by the term "lambda" because I couldn't really understand what it really means. In this context lambda is a function, the single Greek letter was just easier to write in a mathematical notation. Every time you hear "lambda" when talking about functional programming just translate it in your mind to "function".*

[5] *Interestingly Java strings are immutable anyway. It's rather interesting to explore the reasons for this treachery, but that would distract us from our goal.*

[6] *Most functional language compilers optimize recursive functions by transforming them to their iterative alternatives whenever possible. This is called a tail call optimization.*

[7] *The opposite isn't always true. While it is sometimes possible to prove that two pieces of code are equivalent, it isn't possible in all situations.*

[8] *This is actually no slower than storing on the stack because once you introduce a garbage collector, memory allocation becomes an O(1) operation.*

Slava Akhmechet
coffeemug@gmail.com

coffeemug
spakhm

Product at Stripe. Built RethinkDB. Computational neuroscience PhD dropout. Dilettante economist, anthropologist, and rationalist.