Lambda Calculus and Functional Programming

How does an understanding of the lambda calculus contribute to a better understanding of functional programming?

Computer Science
May 2019
Word Count: 3594 words

Table of Contents

Introduction

In 1936, Alonzo Church invented the lambda calculus, introducing "a way of formalizing the concept of effective computability."(Rojas 1) Known as "the smallest universal programming language of the world,"(Rojas 1) the lambda calculus was created in order to "investigate function definition, function application, and recursion in mathematical logic and computer science [...] and [it] forms the basis of a paradigm of computer programming called functional programming" (Bassiri 47). Beginning in 1960 with Lisp, programming languages have been developed that target the functional paradigm, such as Haskell, OCaml, and Erlang.

In their article, "Lambda Calculus and Functional Programming", Anahid Bassiri and Malek Amirian make the claim that these "modern functional languages can be viewed as embellishments to the lambda calculus" (Bassiri 47). However, although they discuss functional programming and the lambda calculus, the authors provide little evidence to show how the two concepts are connected. Therefore, I will be expanding on this claim by exploring the connection between functional programming and the lambda calculus. Specifically, this essay will attempt to answer the question, “How does an understanding of the lambda calculus contribute to a better understanding of functional programming?” First, I will give a brief description of the basics of the lambda calculus in order to give the reader a basic knowledge. Next, I will discuss how pattern matching can be found in the lambda calculus through an analysis of boolean logic in the lambda calculus. Finally, I will investigate referential transparency, a quality found in functional programming, and its connection with intentional equivalence. Through this exploration, I will reveal how the strong correlation between functional concepts and the lambda calculus make a study of lambda calculus beneficial to a functional programmer.

Lambda Calculus

The basis of the lambda calculus is lambda functions, which have the syntax:

```
λ <identifier> . <expression>
```

Lambda functions are similar to the functions found in math, except they have no identifying name and can only take one argument. So the function

```
f(x) = 2x + 1
```

would translate to

```
λx.2x+1
```

in the lambda calculus. To add multiple arguments, functions are nested. For example,

```
f(x, y) = x + y
```

would translate to

```
λx.λy.x+y
```

In order to clean up the syntax for the purposes of this paper, nested functions like the one above will have their arguments condensed, with each letter representing one argument. Therefore, the above lambda expression would be written as

```
λxy.x+y
```

In order to use these functions, one must *apply* arguments to those functions. In mathematics, the argument `1` is applied to the function `f(x)` as `f(1)`. This has the effect of replacing every instance of `x` in `f(x)` with `1`. In the lambda calculus, arguments are applied by putting the argument after the function. For example,

```
(λx.x+1)3
```

applies the argument `3` to `λx.x+1`. This can then be reduced replacing every instance of `x` with `3` and remove the head (`λx`):

```
(λx.x+1)3 ▹ (3)+1
```

If the parameter never appears in the function, then the head is simply removed.

```
(λx.1)3 ▹ 1
```

When an argument is applied to a set of nested lambda functions, it is applied

to the outermost function. Therefore,

```
(λxy.x+y)1 ▹ λy.1+y
```

However, arguments can be reduced in any order.

```
((λxy.x+y)1)2 ▹ (λy.1+y)2 ▹ 1+2
```

and

```
((λxy.x+y)1)2 ▹ (λx.x+2)1 ▹ 1+2
```

reduce to the same expression.

As a side note, "an expression can also be surrounded with parenthesis for clarity" without changing the meaning of the expression (Bassiri 49). In other words, given an expression `E`, one could also write `(E)` to mean the same thing. However, in order to keep expressions clean and parentheses at a minimum, this paper will use the convention that function application is left-associative. In other words, given expressions `E`, `F`, `G`, and `H`,

```
EFGH
```

is the same as

```
(((EF)G)H)
```

Functional Programming Application

Pattern Matching

Boolean Logic

In the lambda calculus, it is possible to represent `true` and `false` with the following functions, `T` and `F`, respectively:

```
T = λxy.x
F = λxy.y
```

(Rojas 6). At first glance, it may not be visible how these represent booleans. However, these in fact encode `if/else` statements. Given some boolean, `b`, and two lambda functions, `t` and `f`, representing what to do if the boolean is true and false respectively, it can be shown how they represent booleans by applying `t` and `f` to `b`:

```
(b)tf
```

If `b` is `true`:

```
(T)tf ▹ (λxy.x)tf ▹ (λy.t)f ▹ t
```

If `b` is `false`:

```
(F)tf ▹ (λxy.y)tf ▹ (λy.y)f ▹ f
```

It is evident that when the boolean is `true`, or `T`, it returns the lambda function `t`, which should be run in the case that the boolean is `true`. On the other hand, when the boolean is `false`, or `F`, it returns the lambda function `f`, which should be run in the case that the boolean is false. This reveals how the lambda functions `T` and `F` encode `if/else` statements.

From Boolean Logic to Pattern Matching

In order to understand how pattern matching works, one must first understand what the sum algebraic data type is. The sum algebraic data type is more commonly known by the names enumeration or disjoint union and will be

referred to as a 'sum type' in this paper. The sum type is a data type that can have multiple variants, but can only be one of those forms at a time (Kuntz). For example, given a `Comparison` sum type with three variants, `Less`, `Equal`, and `Greater`, one could have a variable of type `Comparison` which could store *one* of the three variants, such as `Less`.

The other part of pattern matching is the match statement. The `match` statement takes a sum type and executes different instructions depending on which variant is given. One example of a `match` statement from a functional language, Rust, given the previous sum type, `Comparison`, follows:

```
match some_variable {
    Comparison::Less => {
        // Code for 'Less' variant
    },
    Comparison::Equal => {
        // Code for 'Equal' variant
    },
    Comparison::Greater => {
        // Code for 'Greater' variant
    }
}
```

("The Match Control Flow Operator").

Though it may not seem obvious at first, booleans can be considered to be a sum type. They have two variants, `true` and `false`, and the `if/else` statements found in most, if not all, programming languages are simply a `match` statement specialized for booleans. The first block of instructions should be run if the boolean variant is `true`, while the second block of instructions should be run if the boolean variant is `false`.

Through this perspective of the boolean as a sum type, it becomes evident how sum types and `match` statements can be represented in the lambda

calculus. The boolean sum type has two variants and the lambda functions take two arguments. Each variant returns a different argument; `T` returns the first argument, while `F` returns the second argument. From this analysis, one can construct their own sum type. Each lambda-encoded variant will have one argument for each sum type variant. Then, each variant will return the argument that it corresponds with. As an example, in order to create a lambda-encoded sum type for the aforementioned `Comparison` sum type, each variant would be created with three arguments, `l`, `g`, `e`, for the `Less`, `Greater`, and `Equal` variants respectively.

```
L = λlge
G = λlge
E = λlge
```

Each variant would then return its corresponding argument.

```
L = λlge.l
G = λlge.g
E = λlge.e
```

This is a simple representation of the sum type in the lambda calculus. To use some `Comparison c`, simply apply three functions to it: one for the `Less` variant, `l`; one for the `Greater` variant, `g`; and one for the Equal variant, `e`.

```
(c)lge
```

In this way, match statements for sum types can be represented in the lambda calculus.

Linked Lists

Lists are a tool that is commonly used in functional programming. In fact, the structure of the first functional programming language, Lisp, is based almost completely around lists (McCarthy).

There are multiple ways to represent lists. The two most common structures that are used are arrays and linked lists. Arrays are generally represented by a

pointer that points to a series of data. Accessing the value at index `i` requires following the pointer to the first value in the list, then moving forward `i` values. This access is very quick, happening in `O(1)` time. They are generally immutable, so adding or removing a value to an array requires copying the entire array into an array that fits the new size. Therefore, adding or removing a value takes `O(n)` time.

The other way that lists are commonly represented is in the form of a linked list. Linked lists are composed from a series of nodes. Each node contains a value, also known as the head, and a pointer to the next node in the list, also known as the tail. Access to values in the list is not as fast as arrays because to access a node, one has to climb through each of the preceding nodes. Therefore, access happens in `O(n)` time. However, because each node has a pointer to the next node, removing or adding only requires changing one or two pointers. Therefore, adding or removing a value takes `O(1)` time.

Linked lists are therefore very useful in functional programming. Because it is fast to remove nodes or simply get a reference to the tail, it is common to see recursive code similar to the following, where `isEmpty()` returns `true` if the `LIST` does not contain a list node:

```
define SUM(LIST):
    if LIST.isEmpty():
        return 0
    else:
        return LIST.value + SUM(LIST.tail)
```

In other words, one value is returned when the end of the list is hit; otherwise, the value stored in the current node is processed and combined with the value returned from operating on the tail of the list.

This idea seems to be encoded into the representation of lists in the lambda calculus. In the lambda calculus, the list `[1, 2, 3]` is represented as

follows:

```
λcn.c 1 (c 2( c 3 n))
```

(Some)

It may not be obvious at first glance how the aforementioned process is encoded. However, the parameter `n` is the value that is returned when the end of the list is hit. The parameter `c` is a function that defines how to combine the value stored in the current node and the value returned by the rest of the list. Therefore, the representation of the `SUM` function in the lambda calculus would be:

```
λl.l plus 0
```

where `plus` takes two arguments and returns their sum (Some). If this function were applied to the list given above, it would reduce as follows:

```
(λl. l plus 0) (λcn.c 1 (c 2 (c 3 n)))
(λcn. c 1 (c 2 (c 3 n))) plus 0
(λn. plus 1 (plus 2 (plus 3 n))) 0
plus 1 (plus 2 (plus 3 0))
plus 1 (plus 2 3)
plus 1 5
6
```

(Some) As the expression is reduced, one can see the pattern of recursion found in linked lists, as it nests `plus` until it reaches the end of the list. This shows why linked lists are preferred over arrays in functional programming.

However, the linked list does not simply reveal how linked lists can be used in recursion. The structure of the linked list also reveals the structure for recursion itself. There are two types of cases that need to be dealt with when one is using recursion, represented by the parameters to the list, `c` and `n`. `n` represents the base case, which occurs when a parameter meets a certain condition. For example, with the list, the condition for the base case is that

the list is empty. This is represented in the `SUM` function above by

```
if LIST.isEmpty():
    return 0
```

In general recursion, there can be more than one base case present.

`C` represents the value that should be returned if none of the base cases can be met. This is where the function calls itself. However, when the function calls itself, it reduces the size of one of the parameters. The `SUM` function does this when it calls itself on the tail of the list, reducing the size of the `LIST` parameter by 1:

```
else:
    return LIST.value + SUM(LIST.tail)
```

Linked lists are important to functional programming. Their representation in memory makes them efficient for recursion, which is a useful tool in functional programming. However, the representation of linked lists in the lambda calculus also has important application. It can be used to derive the structure for recursion, that is, return a value for base cases or reduce the size of one of the arguments and call itself. Through these ideas, one can see the importance of linked lists to functional programming.

Referential Transparency

In functional programming, functions are defined as either pure or impure. Functions that are pure are expected to have the property of referential transparency. Referential transparency "indicates that you can determine the result of applying that function only by looking at the values of its arguments"(Mittag), or more generally, "that some given expression always evaluates to the same result in any context"("Referential Transparency").

This concept is found in the non-extensional property of the lambda calculus. In order to describe this property, one must understand the difference between functions in mathematics versus functions in the lambda calculus. In

mathematics, specifically set theory, functions can be considered to be "a set of argument-value pairs"(Alama). They map arguments to values. For example,

```
f(x) = x + 1
```

would map integers to the set

```
{... (-1, 0), (0, 1), (1, 2), ...}
```

where the first value of each pair is the argument `x` and the second value is the resulting value of `f(x)`. If `x` is `0`, `f(x)` maps to `1`. Therefore, functions can be treated as equal if they map to the same set. This means that functions can be manipulated in order to create new functions that are equal to the original function. For example the following two functions are equal because they map to the same sets:

```
f(x) = x + 1
g(x) = x + (2 - 1)
```

This kind of equivalence is *extensional*. Because `2 - 1 = 1` in the actual world, the functions both map to the same sets and are therefore equal.

However, this is only true in worlds where `2 - 1 = 1`. In the lambda calculus, equality is defined as *intentional*. This means that two functions are the same only if they result in the same output in *every* world. Therefore,

```
λx.x+1 ≠ λx.x+(2-1)
```

because if each were reduced in a world where `2 - 1 = 3`, they would be reduced to different functions (Alama):

```
λx.x+(2-1) ▹ λx.x+3 ≠ λx.x+1
```

From this definition of intentional equivalence, it becomes clear that the intentional equivalence of functions is important to the idea of referential transparency. For a pure function to have referential transparency, its results cannot be influenced by external 'global' factors because the results must be determined based on the function arguments alone. The function must

produce the same output in every 'world', therefore implying that the function is required to be intentionally equivalent to itself at all times.

Due to the requirement of intentional equivalency, pure functions cannot be allowed to access mutable global variables. In other words, variables that are outside the scope of a function that can be changed cannot be used inside a pure function. For example, in the following pseudocode, `COUNT` is an impure function because it accesses an external variable `C`.

```
C=0

define COUNT():
    C=C+1
    return C
```

Meanwhile, `ADD_ONE` is pure because it only uses `I`, the argument passed to it.

```
define ADD_ONE(I):
    return I+1
```

It should be noted that any variables that are immutable can be used inside a pure function because they do not change. For example, `ADD_A` and `ADD_TWO` are intentionally equivalent as long as `A = 2` because every instance of `A` could be replaced with the value of `A`.

```
constant A=2

define ADD_A(I):
    return I+A

define ADD_TWO(I):
    return I+2
```

One major implication of the lack of global variable access is that a pure function cannot perform I/O operations. This is because I/O operations read

external variables set by the environment in which the program is run. This causes the result of I/O operations to be different depending on the 'world', or environment, that the operations are being performed in. Therefore, pure functions cannot perform I/O operations.

Conclusion

The purpose of this paper was to determine if there exists a strong correlation between functional programming and the lambda calculus in order to show the benefits of studying lambda calculus for a functional programmer. First, this paper showed how pattern matching, a tool often used by functional programming, can be implemented in the lambda calculus. Next, it discussed the representation of linked lists in the lambda calculus and revealed how its structure reveals the structure of recursion, a functional pattern. Finally, this paper explored the connection between referential transparency and intentional equivalence. In doing so, it came to the conclusion that there is a strong correlation between functional programming and the lambda calculus.

It should be noted that this investigation does not claim that every concept in functional programming was derived from the lambda calculus. It is possible that some concepts have been derived from a different source and cannot be derived in the lambda calculus. Also, even with concepts that can be derived, this investigation does not claim that those concepts were originally from the lambda calculus. For example, pattern matching is a functional programming concept that this investigation then reasoned out in the lambda calculus.

However, this simply shows how closely functional programming connects to the lambda calculus. It shows that the core ideas are similar, so other, more specific concepts that are developed can appear similar, allowing one to reason about functional programming using the basis of the lambda calculus. This reveals how important it is for a functional programmer to understand the lambda calculus. Understanding the lambda calculus can help one to better understand the functional programming paradigm and will therefore help one to better apply it's concepts.

Source Analysis

The source that was used in order to form the basis of knowledge of lambda calculus for this paper was the entry about the lambda calculus by Stanford. This source is a trustworthy source because it is endorsed by Stanford, a university known for its strong Computer Science program. It also cites credible sources, including a paper written by Alonzo Church, the creator of the lambda calculus; a book written by J. Roger Hindley, who helped create a type system based on the lambda calculus; and a paper by John McCarthy, who created the first functional programming language, Lisp. Therefore, this source is a credible source.

This paper also used some primary sources in order to provide reference to real-life languages. In order to provide a reference to Rust, this paper referenced *The Rust Programming Language*, a book written by those who develop the programming language in order to teach newcomers how to program with Rust. To provide reference to Lisp, this paper referenced an essay written by John McCarthy, the creator of Lisp. This essay laid out the basic structure of Lisp. To incorporate Haskell, this paper used a definition from the Haskell wiki, which is maintained by developers of Haskell. Each of these sources were written by those who developed the language. Therefore, they are credible sources for details about their languages.

In order to include global perspective, this paper also utilized an article from the *Global Journal of Researches in Engineering* (GJRE) and an intro to the lambda calculus written by a student at the Free University of Berlin. GJRE is a credible source. It accepts research papers from all over the world, but it also has standards that the research paper must meet before it is published, including a requirement that all research should be peer-reviewed for quality and authenticity. The intro to the lambda calculus is also credible because it was published by the Free University of Berlin, a university well known for its research.

Other sources used by this paper were less credible, though they still proved useful. The two sources from the Stack Exchange Network, the source from Stack Overflow and the source from Software Engineering Stack Exchange, were both questions posted and discussed by those who are members. Anyone

can create an account and add, so it’s not very selective about who can contribute. However, this essay mainly uses these sources in order to get ideas that it then proves. Therefore, these sources were used effectively in order to generate ideas for the paper.

Further Research

One question that this paper didn’t answer was how functional programming developed from the lambda calculus. To answer this question, one would have to look at the history of functional programming and find out when concepts of functional programming were introduced. One could also look at the development from language to language, or at the development of a single functional programming language. One language that would work well for this study would be Haskell, a language developed for the purpose of exploring functional programming.

This paper also didn’t compare the lambda calculus to its counterpart, the Turing Machine. If one were to investigate this topic, one could compare the development of functional programming languages, which are based on the lambda calculus, to the development of imperative programming languages, which are based on the Turing Machine. One could also compare the positive and negative aspects of each. This topic would be a really interesting topic to study.

Works Cited

Alama, Jesse, and Johannes Korbmacher. “The Lambda Calculus.” *Stanford Encyclopedia of Philosophy*, Stanford University, 21 Mar. 2018, plato.stanford.edu/entries/lambda-calculus/.

Bassiri, Anahid Mohammed Reza, and Malek Pouria Amirian. “Lambda Calculus and Functional Programming.” *Global Journal of Researches in Engineering*, vol. 10, no. 2, June 2010, pp. 47–54., globaljournals.org/GJRE_Volume10/gjre_vol10_issue2_7.pdf.

Kuntz, David, et al. “LambdaCast 13: ADTs.” *SoundCloud*, soundcloud.com/lambda-cast/13-adts.

McCarthy, John. *Recursive Functions of Symbolic Expressions and Their Computations by Machine, Part I.* Massachusetts Institute of Technology, Apr. 1960, web.archive.org/web/20131004232653/http://www-formal.stanford.edu/jmc/recursive.pdf.

Mittag, Jörg W, et al. “What Is Referential Transparency?” *Software Engineering Stack Exchange*, softwareengineering.stackexchange.com/questions/254304/what-is-referential-transparency.

“Referential Transparency.” *Haskell in Industry - HaskellWiki*, wiki.haskell.org/Referential_transparency.

Rojas, Raúl. “A Tutorial Introduction to the Lambda Calculus.” *Freie Universität Berlin*, www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf.

Some, and Gallais. “Sum of List Elements and Length of List in Lambda Calculus.” Stack Overflow, stackoverflow.com/questions/46186358/sum-of-list-elements-and-length-of-list-in-lambda-calculus.

"The Match Control Flow Operator." *The Rust Programming Language*, doc.rust-lang.org/book/ch06-02-match.html.