# Lambda Calculus as a Programming Language

Jeffrey Lin

May 11, 2019

## 1 Introduction

The Lambda Calculus was invented by the mathematician Alonzo Church in the 1930s around the same time Alan Turing came up with the idea of the Turing Machine. It was later proved that the Lambda Calculus and the Turing Machine are equivalent: whatever computation one can represent, the other can as well. While the Turing Machine is the basis for imperative languages and their constructs today, the Lambda Calculus is the basis of functional programming.

Note that for the purposes of our model, we define the Lambda Calculus using Harrison's *Introduction to Functional Programming*.

## 2 Defining and Manipulating Terms

There are three categories of terms in the Lambda Calculus: variables, applications, and abstractions. In OCaml, we can therefore represent the type term as:

```
type term = Var of string | Abs of string * term | App of term * term
```

Using this type, we will model computation as a tree, which can be easily traversed and manipulated with recursive pattern matching functions.

### 2.1 Printing

To pretty-print terms, we will want to use as few parentheses as possible. An initial design for the term printer `string_of_term` always used parentheses around both terms in applications `App(s, t)`. However, this resulted in nested applications producing too many parentheses, which made many complex terms look ugly. In the latest version, we use a mutually recursive set of functions `string_of_term` and `string_of_exp` which will only output a pair of parentheses when the term inside is not a simple variable.

### 2.2 Free and Bound Variables

As the definition of free and bound variables is given in terms of mathematical sets, we construct a module set `VarSet` of `String`s that will allow us to perform these set operations.

Free variables are those that are excluded from the scope of an abstraction, whereas bound variables are those that are included in the scope of an abstraction. While this isn't immediately useful, we will need to know what the free variables of a given term are when we model substitution.

### 2.3 Substitution

It is interesting to note that abstractions and applications are actually the inverses of each another. We write the `substitute` function based on a few basic rules:

- To perform substitution on variables, replace the "old" term with the "new" term.
- We can perform substitution on applications by "propagating" the substitution inwards to both subterms.
- Substitution on an abstraction is a bit trickier. We must be careful to not accidentally bind or free any variables. Given a term $\lambda$ `x.s`, we can safely perform substitution if the term being replaced is not a free variable in $s$, or $x$ is not a free variable in the replacing term. Otherwise, we have to rename $x$ to a variable that is not free in either $s$ or the replacing term, after which we can proceed as normal.

To help us to choose a "safe" variable, we perform the proper operations and pass the `VarSet` of free variables to `fresh_var`, which returns a variable that we can rename $x$ to.

### 2.4 Conversions

There are three types of conversions: alpha ($\alpha$), beta ($\beta$), and ($\eta$). We actually do an implicit alpha conversion in the last case for substitutions, when we rename $x$ before proceeding with the substitution. The most important of these is the beta conversion, in which we can take an application of the form `(`$\lambda$ `x.s) t` and substitute $t$ for $x$ in $s$. Eta conversion is interesting for logical analysis as it can assist us with potentially reducing even further, but isn't as useful as the beta reduction for simulating programs.

### 2.5 Normalization of Terms

It can be proved that if we continuously process the expression by reducing the "leftmost outermost" term of the form `(`$\lambda$ `x.s) t`, we will eventually end up with an expression that can no longer be reduced, i.e. the expression will be in normal form. Given a term as input, `normalize_step` returns the term that is the result of one step of reduction. To fully reduce, `normalize` runs `normalize_step` until the output is the same as the input, with the assumption being that no more reductions are possible after that point.

## 3 Executing Programs

We can now write and reduce arbitrary terms, but that is not very entertaining or useful to the typical end user. While the Lambda Calculus can model any computation, it does not do so in a manner that is easy to read.

## 3.1 Defining Procedures

To make terms slightly easier to work with, we define a simple `procedure` type that allows us to alias terms to human-readable identifiers:

```
type procedure = Def of string * Terms.term | Use of Terms.term | Noop | Eof
```

A `procedure` corresponds to a single statement in our language. We can use a `Def` to associate an identifier with the term it represents, and a `Use` to perform operations (the most computationally useful one being reduction/normalization). `Noop` and `Eof` are more for the implementation, as some statements, such as comments, may not be of value to our interpreter.

## 3.2 Defining Programs

Now that the concept of procedures is well-defined, we can combine sequences of them to create larger programs. We split the definition of a program into two parts: its environment and routines:

```
module EnvMap = Map.Make (struct
  type t = Terms.term
  let compare = compare
end)

type program = { env: string EnvMap.t ; routine: Terms.term list }
```

The environment `env` will store `Def`s and allow us to look up terms by their identifiers. We use a list of `term`s to represent the terms that we actually want to see the normalization of. In reality, this list could be rather short in comparison to the size of the environment.