Preface: A Tale of Two Theories

The Theory

In the middle of the nineteen-thirties, two men were laying the foundations of computation. It was in the year of nineteen-thirty-six that each released his *magnum opus*. Alonzo Church revealed to the world a formal system of computation expressed by function definition and evaluation, and Alan Turing shared his vision of a hypothetical device, one which could simulate the logic of any algorithm. In retrospect, the two works provided equivalent foundation for this brave new world of computers.

The Lambda Calculus

The language developed by Church was one of very simple syntax. All expressions in the language were one of the following three types.

- A function definition of the form λab where a is a variable and b is an expression.
- A function evaluation of the form $(a)b$ where a is a function and b an expression.
- A variable reference.

This can be summarized using Backus-Naur Form as follows. An expression is said to be replaceable by one of the values separated by a vertical pipe; each of these can then in turn be reduced by referring again to the definition of `<expr>`.

$$
\begin{aligned}
\text{<expr>} \quad ::= \; & \lambda\text{<var><expr>} \\
| \; & (\text{<expr>})\text{<expr>} \\
| \; & \text{<var>}
\end{aligned}
$$

As far as the semantic meaning of this language, there is only one operation. Any expression of the second form (`2.`) can be reduced by substituting the passed value for instances of the argument in the function body.

The elegance and minimalism of this language is striking. Despite its appearance, this language is capable of universal computation; shis tiny language is sufficient for expression of any algorithm, i.e., any concept or thought. This language, however simple, can become tedious with the heavy nesting of expressions and very difficult to actually evaluate to something meaningful. A more practical outlook is necessary when dealing with implementation by a machine.

Turing Machines

Turing provided us with this different outlook, one of simplicity in implementation rather than expression. Turing presented a machine capable of shifting a tape, modifying values, and maintaining state, following a set of prescribed rules in its performance of these operations. A single rule is of the following form.

```
<state> ::= A | B | …
<value> ::= 0 | 1
<direction> ::= R | L
<rule> ::= <state> <value> <value> <direction> <state>
```

For a given rule, the first two values, *state* and *value*, serve to identify whether a rule is applicable, and the following values describe the manipulations that should take place. A Turing ruleset would then be a sequence of rule expressions. A certain Turing Machine has a given ruleset to achieve a specific goal. Perhaps this goal is to find the roots of a polynomial, or add two bits. The bit adding example is very easily implemented in simple rules, but any more difficult task can become quite complex.

Abstraction

The goal of this book is to build upon these rudimentary definitions of computation to reach a level of abstraction at which the fruits of your programming become clear and expressive. We ascend from these mess-making foundations, exiting Turing's tarpit, with the aid of abstraction over fundamentals.

When you add abstraction, you are simply shedding details which make a concept hard to interpret in its entirety. Hence throughout this book, we will be shifting focus from Lambda Calculus functions, to a more symbolic language, then all the way back down to Turing Machines. Our journey will be all across the board, but the goal is for the reader to understand traversal between any level of abstraction, amongst which are:

- Representing a Symbolic Language with Functional Primitives
- Simulating a Turing Machine in a Symbolic Language
- Interpreting a Functional Language
- Interpreting a Language in Itself
- Converting a Symbolic Language to Manipulations of a Register Machine

All of the above will be addressed directly, and other traversals will become apparent given this context.

by Matt Neary