Chapter 1

λ-Calculus

The λ-calculus is one of the first formal attempts to define computation, doing so through the concept of a function. An intuitive definition of a function might be: a mathematical object that takes an input and returns an output—simple yet capturing the abstract nature of functions. Depending on the context, different formal systems can be used to describe what a function does and, in some cases, how it does it.

This distinction between what a function does and how it does it is a crucial aspect of computation. Understanding when two functions can be considered equivalent is fundamental to the theory of computation, as it allows us to reason about different implementations of the same underlying behavior. Prior to the development of lambda calculus, the equivalence of functions was understood in terms of the equivalence of input-output pairs. In other words, if two functions that share the same domain, f and g, return the same output for every possible input, then f and g are considered equivalent. The λ-calculus extends this notion of equivalence, known as denotational equivalence, by providing a deeper insight into how functions behave. It introduces the concept of operational equivalence—the idea that two functions are equivalent if they operate in the same step by step, that is, if they exhibit the same computational behavior or reduce in the same way during evaluation.

As a historical note and to give credit where credit is due, the lambda calculus was invented by Alonzo Church and Stephen Kleene during the 1930s. Since its inception, it has been used to explore the limits of computability and decidability, establish the core principles of the functional programming paradigm, and advance the study of formal systems and proof theory.

Untyped λ-Calculus

In order to gain a better understanding of what the λ-calculus is all about, let us develop a simple example to familiarize ourselves before we begin with a more formal approach to this discipline. Consider the function $f(x) = x + 1$, the most straightforward way to express this using λ-calculus would be $(\lambda x.x + 1)$, [1] where the lambda denotes that x is being captured and used as a parameter to perform some computation. Evaluating $f(2)$ using this newly created lambda term would look like this: $(\lambda x.x + 1)(2) \rightsquigarrow (2 + 1) \rightsquigarrow 3$. This process of reducing a λ-expression is referred to as β reduction.

[1] This example abuses notation in the pursuit of understanding, the λ-calculus only allows expressions of the form defined in 1.0.1

Another concept that might be worth-while glancing over, is what in the jargon is referred to as first-class citizenship, or more formally high-order. A λ-term being high-order, refers to the idea that terms (functions) are treated indistinguisably from terms (arguments). See what I did there, yes, that is exactly what high-order means. One can intuitively apreciate the computational expresivenes this brings with it, nevertheless, let us show how this can be useful: Consider now the reduction procedure of the following lambda term:

$$\begin{aligned}\overbrace{(\lambda f.\lambda y.f\,f\,y)}^{A}\overbrace{(\lambda x.x+1)}^{B}\overbrace{(2)}^{C} &\rightsquigarrow (\lambda y.(\lambda x.x+1)(\lambda x.x+1)y)(2)\\ &\rightsquigarrow (\lambda x.x+1)(\lambda x.x+1)(2)\\ &\rightsquigarrow (\lambda x.x+1)(3)\\ &\rightsquigarrow (4)\end{aligned}$$

If we take a close look, at A, we notice the term applies f to x twice, and that as a consequence, B is being applied to C twice. And so, very easily, we have implemented the $x+2$ function by extending our definition for $x+1$.

Lambda Terms

Having established a basic intuition behind the lambda calculus, let us define $\mathbb{V} = \{x, y, z, ...\}$, as the set of all possible variable names, this set is of little real importance, but a must-have in order to formalize some of the upcoming definitions.

Definition 1.0.1. (Λ; The Set of all λ-terms): The language for expressions in the lambda calculus can be defined using a grammar in BNF notation like this:

$$M = x \mid MM \mid \lambda x.M \quad x \in \mathbb{V}$$

Example 1.0.1. Some examples of valid λ terms that belong to Λ: y, $\lambda x.xx$, $\lambda x.\lambda y.x$, $(\lambda x.xx)(\lambda y.y)z$.

Definition 1.0.2. (Sub; Multiset of Subterms): Since every lambda term is made up of other smaller lambda terms, it is only natural to define the set of subterms:

$$\begin{aligned}&\mathbf{Sub}(x) = \{x\}\\ &\mathbf{Sub}(MN) = \mathbf{Sub}(M) \cup \mathbf{Sub}(N) \cup \{MN\}\\ &\mathbf{Sub}(\lambda x.M) = \mathbf{Sub}(M) \cup \{\lambda x.M\}\end{aligned}$$

Where $x \in \mathbb{V}$ and $M, N \in \Lambda$.

Definition 1.0.3. (Proper Subterm): A term $M \in \mathbf{Sub}(M),\ M \in \Lambda$ is said to be proper if $S \neq M$. Thus, $\mathbf{Sub}(M) \setminus \{M\}$ would be the multiset of proper subterms.

Definition 1.0.4. (Var; The set of Variables in a lambda term):

$$\begin{aligned}&\mathbf{FV}(x) = x\\ &\mathbf{FV}(MN) = \mathbf{FV}(M) \cup \mathbf{FV}(N)\\ &\mathbf{FV}(\lambda x.M) = \mathbf{FV}(M) \setminus \{x\}\end{aligned}$$

Definition 1.0.5. (FV; The set of Free Variables in a lambda term): Variables that are not captured by means of an abstraction are said to be free, non-free variables are said to be bound.

$$\begin{aligned}
&\mathbf{FV}(x) = x \\
&\mathbf{FV}(MN) = \mathbf{FV}(M) \cup \mathbf{FV}(N) \\
&\mathbf{FV}(\lambda x.M) = \mathbf{FV}(M) \setminus \{x\}
\end{aligned}$$

Example 1.0.2. Compute the free variables of $\lambda x.\lambda y.yxz$:

$$\begin{aligned}
\mathbf{FV}(\lambda x.\lambda y.yxz) &= \mathbf{FV}(\lambda y.yxz)) \setminus \{x\} \\
&= \mathbf{FV}(yxz) \setminus \{x, y\} \\
&= \{x, y, z\} \setminus \{x, y\} \\
&= \{z\}
\end{aligned}$$

Of course, z is the only free variable in the expression as it is the only variable that is not captured by some lambda abstraction.

Definition 1.0.6. A term $M \in \Lambda$ is closed if and only if $\mathbf{FV}(M) = \emptyset$, closed lambda terms are often referred to as combinators. Λ^0 is the set of all closed lambda terms.

Fixed Point Combinators and Recursion

Simply Typed λ-Calculus

The Untyped Lambda Calculus in computationally equivalent to a Turing machine. However, with great computational power comes limited decidability of properties, leading to non-termination, or expressions such as $x(\lambda y.y)$, whose meaning is unclear.

A classic example of non-termination is the Y combinator, however, the Simply Typed Lambda Calculus does not allow such expressions, as its type system is unable to assign a valid type to them.

To understand why, consider the role of function types: in the world of functions, a function maps values from a domain to a range. The Simply Typed Lambda Calculus enforces this structure explicitly, ensuring that every function application is well-typed and preventing self-application patterns that would lead to paradoxes or infinite loops.

Having grasped the untyped lambda calculus's Turing completeness and ability to compute all computable functions, we now seek properties related to decidability. To this end, we introduce the simply typed lambda calculus. Although it possesses less computational power than its untyped counterpart, it offers attractive features regarding decidability that will be useful later on.

Chapter 2

The Curry-Howard Correspondence

A Primer on Logic