Modern Computation

A Unified Approach

S. S. Chandra
Ph.D
May 2019

Draft
Version: 0.26

“Computing Science is no more about computers
than Astronomy is about telescopes.”

Edsger W. Dijkstra (1930-2002)

CHAPTER 5

λ-Calculus

"The purpose of computing is insight, not numbers."

Richard Hamming (1915-1998)

In a previous chapter on Turing machines, we saw that Alan Turing tried to conceptualise the simplest form of a computer. He envisioned replacing a human with an electro-mechanical read/write head, some instructions and a memory for intermediate results that also kept track of the states of computation. This state-based approach serves us well in providing an intuitive understanding of a computer and computation. We also saw that the instructions and indeed the machine itself were a form of code, just as Gödel thought of theorems as unique integers that encoded all of mathematics.

In this chapter, we will describe an alternative approach that will show that all of computation can be deconstructed and derived from just a single concept. One simple concept that is required to construct all necessary logic, loops, numbers and recursion necessary to create a system that is still Turing complete. In much the same way that Turing 'built' an abstract encoding machine that represented the simplest computer, we will follow in the footsteps of Alonzo Church [1932], who constructed an abstract encoding scheme that can represent computation itself, without any assumption of a physical machine, based on the concept of a function. It allows us to solve computational problems with simpler language and notation than Turing machines, all the while better supporting parallel computations, high-level programming and being equivalent in computing power to Turing machines. This universal theory of computation based on functions is called λ-Calculus [Church, 1932], where the symbol λ is the Greek symbol lambda.

5.1 Functions

Recall that in our mathematics chapter (see section 2.3) we defined a function as a mapping between a domain and a range through the use of variables. For example, a function $\boldsymbol{f}$ repres-

enting the quadratic mapping of variable $\boldsymbol{x}$ can be written as

$$\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{x}^2 \tag{5.1}$$

where the $\boldsymbol{x}$ in the parentheses is the argument to the function $\boldsymbol{f}$ and the function body defined after the equal sign. The notation above is not as compact and succinct as possible for encoding computations. We would essentially like to be able to write programs based on functions and having extraneous parentheses and equal signs will lengthen and complicate the our encodings.

Church [1932] introduced the λ notation to functions to allow long strings of functions to be chained or composed together that is both minimal and compact. We remove the parentheses and equal signs because they are implied, introduce the λ symbol to denote the beginning of a function and use the period symbol to denote the chaining or composition of all elements. For example, the simplest expression is $\lambda\boldsymbol{x}.\ \boldsymbol{x}$, where λ denotes function definition with arguments and the symbols after the period as the body of the function. Here we replace $\boldsymbol{x}$ with itself and effectively do nothing to $\boldsymbol{x}$. We can define this as the identity function

$$I := \lambda\boldsymbol{x}.\ \boldsymbol{x} \tag{5.2}$$

Another example is the quadratic mapping above that can be represented as the following in his notation

$$\lambda\boldsymbol{x}.\ \boldsymbol{x}^2 \tag{5.3}$$

The functions defined in this way are ***always*** unary, i.e. only take one argument. We can then handle multiple arguments as in the following example of addition of the two variables $\boldsymbol{x}$ and $\boldsymbol{y}$ by nesting functions

$$\lambda\boldsymbol{x}.\ \lambda\boldsymbol{y}.\ \boldsymbol{x} + \boldsymbol{y} \tag{5.4}$$

5.1.1 Reductions and Normal Forms

This nesting has an operator precedence in which arguments, i.e. the terms with the λ symbol, are processed from left to right. The terms following the period after the last argument denotes the function bodies, which as simplified from right to left one at a time. For example, this can be seen in equation (5.4) as indicated by parentheses for clarity $(\lambda\boldsymbol{x}.\ (\lambda\boldsymbol{y}.\ \boldsymbol{x} + \boldsymbol{y}))$. If we wish to apply this computation as $\boldsymbol{x} = 2$ and $\boldsymbol{y} = 3$, we would write

$$\begin{aligned} (\lambda\boldsymbol{x}.\ (\lambda\boldsymbol{y}.\ \boldsymbol{x} + \boldsymbol{y})3)2 &= (\lambda\boldsymbol{y}.\ 2 + \boldsymbol{y})3 \\ &= 2 + 3 \\ &= 5 \end{aligned} \tag{5.5}$$

The process of simplification of λ-expression is called β-reduction. where the symbol β is the Greek symbol beta. The final result of a β-reduction that cannot be simplified further is called the β-normal form. We need not only require numbers to simplify expressions. For example, the following expressions can be β-reduced as well

$$(\lambda\boldsymbol{x}.\ \boldsymbol{x}\boldsymbol{x})(\lambda\boldsymbol{y}.\ \boldsymbol{y}) = (\lambda\boldsymbol{y}.\ \boldsymbol{y})(\lambda\boldsymbol{y}.\ \boldsymbol{y}) \tag{5.6}$$

as well as

$$(\lambda \boldsymbol{x}.\ \boldsymbol{x}\ (\lambda \boldsymbol{x}.\ \boldsymbol{x}))\boldsymbol{y} = \boldsymbol{y}(\lambda \boldsymbol{x}.\ \boldsymbol{x}) \tag{5.7}$$

The only trap that we need to be careful of is the difference between what variables are bound to the function (are related to the arguments of the function) and what variables are free from them (do not appear in the arguments and appear from "nowhere"). Consider the $\boldsymbol{y}$ in the example

$$(\lambda \boldsymbol{x}.\ \boldsymbol{x}\boldsymbol{y}) \tag{5.8}$$

The $\boldsymbol{y}$ is not input to the function via arguments and is thus a ***free variable***, in contrast to $\boldsymbol{x}$ that is a ***bound variable***. It is therefore quite easy to mix up free and bound variables of the same name. For example, the expression below has $\boldsymbol{y}$ in two different contexts

$$(\lambda \boldsymbol{x}.\ (\lambda \boldsymbol{y}.\ \boldsymbol{x}\boldsymbol{y}))\boldsymbol{y} \tag{5.9}$$

The $\boldsymbol{y}$ on the far right is free but the $\boldsymbol{y}$ in the parentheses is bound. In these cases, we are able to rename variables, since the variables within functions are dummy variables and have no physical meaning other than notation. In our example, we can rename the $\boldsymbol{y}$ in the function as $\boldsymbol{t}$ to get

$$(\lambda \boldsymbol{x}.\ (\lambda \boldsymbol{t}.\ \boldsymbol{x}\boldsymbol{t}))\boldsymbol{y} \tag{5.10}$$

which results in the following β-normal form

$$(\lambda \boldsymbol{t}.\ \boldsymbol{y}\boldsymbol{t}) \tag{5.11}$$

Sometimes, the notation can be made more compact by concatenating the λ arguments since it is also implied, so that the equation (5.4) reduces to the following

$$\lambda \boldsymbol{x}\boldsymbol{y}.\ \boldsymbol{x} + \boldsymbol{y}$$

This is an example of Currying, in honour of Haskell Curry who formalised the notation of λ-Calculus [Curry and Feys, 1958]. For simplicity and easier interpretation, we will keep the notation more verbose as in equation (5.4) throughout this book.

5.1.2 Pure Functions

We also make one more requirement of the functions when using them within λ-Calculus; the functions must be pure, i.e. they have no internal state and no side effect(s). This means that functions only take inputs and produce outputs as described by the notation and do not change the inputs and any other variables or produce additional output, such as output to a terminal screen. In other words, operations such as 'print' are not allowed as it changes the state of 'outside' function elements. Though not allowed in λ-Calculus, programming languages that support λ-Calculus may still allow these operations for practical reasons. This has the significant benefit of supporting massively parallel computations, since unsolicited memory/hardware

accesses such as write operations, are kept to a minimum to reduce the chance of simultaneous access that can cause programs to crash and ensures that operations are always well defined.

However, the λ-Calculus as we have defined it so far only has the concept of a function inside it. There are no numbers, or loops or even numbers! In the next section, we will define a way to encode all that is required to make a Turing complete computation system.

5.2 Church Encoding

The main challenge that λ-Calculus initially has is that it is completely bare, only supporting the concept of a unary, pure functions and nothing else, not even numbers or concepts of true and false. Though we can handle multiple arguments through Currying, we need to build the remaining concepts into the calculus from the ground up. Church [1932] developed an encoding scheme to construct all the necessary elements to create a Turing complete system, though Turing was developing his Turing machines independently at the same time. This process of encoding functionality into the notation of λ-Calculus is now known as Church encoding. We will not only show that the entire Turing complete functionality is possible, but also give examples of how elegant and simple it is to build multipliers and self interpreters within λ-Calculus.

5.2.1 Church Numerals

The first step is to create the concept of numbers within λ-Calculus and then define a method of counting between them. Numbers can be encoded by simply applying a function repeatedly $\boldsymbol{n}$ times depending on the number $\boldsymbol{n}$ being represented. For example, we can represent the number 1 as a function $\mathcal{N}_1$ as taking an input function $\boldsymbol{f}$ and variable $\boldsymbol{x}$ and applying $\boldsymbol{f}$ exactly once to $\boldsymbol{x}$ as

$$\mathcal{N}_1 := \lambda \boldsymbol{f}.\ \lambda \boldsymbol{x}.\ \boldsymbol{f}(\boldsymbol{x}) \tag{5.12}$$

Likewise for other numbers 2 and 3 as

$$\mathcal{N}_2 := \lambda \boldsymbol{f}.\ \lambda \boldsymbol{x}.\ \boldsymbol{f}(\boldsymbol{f}(\boldsymbol{x})) \tag{5.13}$$

$$\mathcal{N}_3 := \lambda \boldsymbol{f}.\ \lambda \boldsymbol{x}.\ \boldsymbol{f}(\boldsymbol{f}(\boldsymbol{f}(\boldsymbol{x}))) \tag{5.14}$$

and so on, noting that we can define $\mathcal{N}_0$ as applying $\boldsymbol{f}$ zero times

$$\mathcal{N}_0 := \lambda \boldsymbol{f}.\ \lambda \boldsymbol{x}.\ \boldsymbol{x} \tag{5.15}$$

The actual form of the function $\boldsymbol{f}$ is not important right now and we can use the identity I as the function by default. These numbers are known as Church numerals.

We can then adopt the concept of the natural numbers $\mathbb{N}$ as defined by Peano and his so called Peano numbers to develop counting within the calculus. In Peano numbers, we define the natural numbers $\mathbb{N}$ recursively as the successor function $\mathcal{S}$, i.e. the "what comes after" function. For example, the successor of 0 is 1, i.e. $\mathcal{S}(0) = 1$, and $\mathcal{S}(1) = 2$ etc. Thus, we can write that $3 = \mathcal{S}(\mathcal{S}(\mathcal{S}(0)))$ and likewise for any number required by using the $\mathcal{S}$ function recursively. Therefore, we can count in the λ-Calculus by developing the successor "program" or function

that takes as input a Church numeral to compute the successor from and the function to use to give us an output Church numeral as

$$\mathcal{S} := \lambda \boldsymbol{n}.\ \lambda \boldsymbol{y}.\ \lambda \boldsymbol{x}.\ \boldsymbol{y}(\boldsymbol{nyx}) \tag{5.16}$$

The successor $\mathcal{S}$ function takes three arguments because we are effectively telling it to apply an additional function $\boldsymbol{y}$ to the numeral that is input to generate the next numeral, which itself requires two arguments. To test, let's apply this to $\mathcal{N}_0$, our numeral for zero. We substitute it into the successor function to find that

$$\begin{aligned} \mathcal{S}_0 &= \mathcal{S}(\lambda \boldsymbol{f}.\ \lambda \boldsymbol{x}.\ \boldsymbol{x}) && (5.17)\\ &= \lambda \boldsymbol{n}.\ \lambda \boldsymbol{y}.\ \lambda \boldsymbol{x}.\ \boldsymbol{y}(\boldsymbol{nyx})(\lambda \boldsymbol{f}.\ \lambda \boldsymbol{x}.\ \boldsymbol{x}) \\ &= \lambda \boldsymbol{y}.\ \lambda \boldsymbol{x}.\ \boldsymbol{y}((\lambda \boldsymbol{f}.\ \lambda \boldsymbol{x}.\ \boldsymbol{x})\boldsymbol{yx}) \\ &= \lambda \boldsymbol{y}.\ \lambda \boldsymbol{x}.\ \boldsymbol{y}((\lambda \boldsymbol{x}.\ \boldsymbol{x})\boldsymbol{x}) \\ &= \lambda \boldsymbol{y}.\ \lambda \boldsymbol{x}.\ \boldsymbol{y}(\boldsymbol{x}) && (5.18) \end{aligned}$$

which is the definition of $\mathcal{N}_1$, since the dummy name $\boldsymbol{y}$ is equivalent to our initial name $\boldsymbol{f}$ in equation (5.12). Computing the successor of $\mathcal{N}_1$, we see the purpose of the $\boldsymbol{y}$ argument in the $\mathcal{S}$ function

$$\begin{aligned} \mathcal{S}_1 &= \mathcal{S}(\lambda \boldsymbol{f}.\ \lambda \boldsymbol{x}.\ \boldsymbol{f}(\boldsymbol{x})) && (5.19)\\ &= \lambda \boldsymbol{n}.\ \lambda \boldsymbol{y}.\ \lambda \boldsymbol{x}.\ \boldsymbol{y}(\boldsymbol{nyx})(\lambda \boldsymbol{f}.\ \lambda \boldsymbol{x}.\ \boldsymbol{f}(\boldsymbol{x})) \\ &= \lambda \boldsymbol{y}.\ \lambda \boldsymbol{x}.\ \boldsymbol{y}((\lambda \boldsymbol{f}.\ \lambda \boldsymbol{x}.\ \boldsymbol{f}(\boldsymbol{x}))\boldsymbol{yx}) \\ &= \lambda \boldsymbol{y}.\ \lambda \boldsymbol{x}.\ \boldsymbol{y}((\lambda \boldsymbol{x}.\ \boldsymbol{y}(\boldsymbol{x}))\boldsymbol{x}) \\ &= \lambda \boldsymbol{y}.\ \lambda \boldsymbol{x}.\ \boldsymbol{y}(\boldsymbol{y}(\boldsymbol{x})) && (5.20) \end{aligned}$$

It is there to rename the function $\boldsymbol{f}$ in the numeral to $\boldsymbol{y}$ in order to apply an additional $\boldsymbol{y}$ to compute the next numeral as we've defined in the calculus.

5.2.2 Arithmetic

Having defined numbers and counting in the calculus, we can now compute arithmetic operations such as add and multiply. In terms of multiplication, we simply use the Church numerals directly, so that multiplication of numerals $\boldsymbol{n}$ and $\boldsymbol{k}$ can be defined as

$$\mathcal{B} := \lambda \boldsymbol{n}.\ \lambda \boldsymbol{k}.\ \lambda \boldsymbol{f}.\ \boldsymbol{n}(\boldsymbol{kf}) \tag{5.21}$$

Then the multiplication of $\mathcal{N}_2$ and $\mathcal{N}_2$ is as follows

$$\begin{aligned} (\lambda \boldsymbol{n}.\ \lambda \boldsymbol{k}.\ \lambda \boldsymbol{f}.\ \boldsymbol{n}(\boldsymbol{kf}))\mathcal{N}_2\mathcal{N}_2 &= (\lambda \boldsymbol{k}.\ \lambda \boldsymbol{f}.\ \mathcal{N}_2(\boldsymbol{kf}))\mathcal{N}_2 && (5.22)\\ &= \lambda \boldsymbol{f}.\ \mathcal{N}_2(\mathcal{N}_2\boldsymbol{f}) \\ &= \lambda \boldsymbol{f}.\ (\lambda \boldsymbol{f}.\ \lambda \boldsymbol{x}.\ \boldsymbol{f}(\boldsymbol{f}(\boldsymbol{x})))((\lambda \boldsymbol{f}.\ \lambda \boldsymbol{x}.\ \boldsymbol{f}(\boldsymbol{f}(\boldsymbol{x})))\boldsymbol{f}) \\ &= \lambda \boldsymbol{f}.\ (\lambda \boldsymbol{f}.\ \lambda \boldsymbol{x}.\ \boldsymbol{f}(\boldsymbol{f}(\boldsymbol{x})))(\lambda \boldsymbol{x}.\ \boldsymbol{f}(\boldsymbol{f}(\boldsymbol{x}))) \\ &= \lambda \boldsymbol{f}.\ \lambda \boldsymbol{x}.\ (\lambda \boldsymbol{x}.\ \boldsymbol{f}(\boldsymbol{f}(\boldsymbol{x})))((\lambda \boldsymbol{x}.\ \boldsymbol{f}(\boldsymbol{f}(\boldsymbol{x})))(\boldsymbol{x})) \\ &= \lambda \boldsymbol{f}.\ \lambda \boldsymbol{x}.\ (\lambda \boldsymbol{x}.\ \boldsymbol{f}(\boldsymbol{f}(\boldsymbol{x})))(\boldsymbol{f}(\boldsymbol{f}(\boldsymbol{x}))) \\ &= \lambda \boldsymbol{f}.\ \lambda \boldsymbol{x}.\ \boldsymbol{f}(\boldsymbol{f}(\boldsymbol{f}(\boldsymbol{f}(\boldsymbol{x})))) && (5.23) \end{aligned}$$

which is $\mathcal{N}_4$ as expected.

In terms of addition with respect to the successor function, we note that adding 2 to 2 is equivalent to computing the successor of 2 two times, which we can write as $2\mathcal{S}_2$ as

$$\begin{aligned} 2\mathcal{S}_2 &= (\lambda \boldsymbol{s}.\ \lambda \boldsymbol{z}.\ \boldsymbol{s}(\boldsymbol{s}(\boldsymbol{z})))\ \mathcal{S}\ (\lambda \boldsymbol{s}.\ \lambda \boldsymbol{x}.\ \boldsymbol{s}(\boldsymbol{s}(\boldsymbol{x}))) && (5.24) \\ &= (\lambda \boldsymbol{z}.\ \mathcal{S}\ (\lambda \boldsymbol{s}.\ \lambda \boldsymbol{x}.\ \boldsymbol{s}(\boldsymbol{s}(\boldsymbol{x})))(\mathcal{S}\ (\lambda \boldsymbol{s}.\ \lambda \boldsymbol{x}.\ \boldsymbol{s}(\boldsymbol{s}(\boldsymbol{x})))(\boldsymbol{z}))) \\ &= (\lambda \boldsymbol{z}.\ \mathcal{S}_2(\mathcal{S}_2(\boldsymbol{z}))) && (5.25) \end{aligned}$$

which we can simplify as we have done for the multiplication example above.

5.2.3 Boolean Algebra

The next step is to create the concepts of true and false, known as Boolean types that forms the basis of Boolean algebra. Often in conditional statements, we need to make a decision based on a value or expression. For example, we may have statement that "If water in my cup is cold, I will do this, but if it is not I will do that". In these expressions, the true part of the if statement comes first, i.e. we choose to do the first task based on if the expression is true and the second part when the expression is false. We can mimic this in λ-Calculus by taking two arguments and defining that if the first is chosen, it represents true and if the second is chosen, then it is false. In other words, we may define true $\mathcal{T}$ and false $\mathcal{F}$ as the following in λ-Calculus

$$\mathcal{T} := \lambda \boldsymbol{x}.\ \lambda \boldsymbol{y}.\ \boldsymbol{x} \quad (5.26)$$

$$\mathcal{F} := \lambda \boldsymbol{x}.\ \lambda \boldsymbol{y}.\ \boldsymbol{y} \quad (5.27)$$

This may seem arbitrary, but Boolean algebra forms a key part in digital computers as binary values are easily modelled after true and false values. We can then perform simple logical operations on Boolean values known as AND, OR and NOT, thus verifying whether our definitions for $\mathcal{T}$ and $\mathcal{F}$ are consistent.

A NOT operation takes a Boolean value and negates it, so that true becomes false and false becomes true. Consider now how we defined true and false, it selects one of the two inputs depending on the Boolean value. The NOT operation toggles the value so we need to select the opposite value based on the current argument. The inputs are Booleans $\boldsymbol{p}$ and $\boldsymbol{q}$, i.e. selector functions, so we can then define the NOT as an opposing selector function

$$\text{NOT} := \lambda \boldsymbol{p}.\ \boldsymbol{p}\mathcal{F}\mathcal{T} \quad (5.28)$$

In the body, we can see the Boolean $\boldsymbol{p}$ will select either $\mathcal{F}$ if it is true and $\mathcal{T}$ if it is false according to our convention for true and false. Likewise, we can define the AND operation as the selector function that requires both $\boldsymbol{p}$ and $\boldsymbol{q}$ are both true, so we can define it as

$$\text{AND} := \lambda \boldsymbol{p}.\ \lambda \boldsymbol{q}.\ \boldsymbol{p}\boldsymbol{q}\mathcal{F} \quad (5.29)$$

In the body, we can see if $\boldsymbol{p}$ is false, return false because both Booleans need to be true and the value of $\boldsymbol{q}$ is not necessary to make the decision, otherwise return the value of $\boldsymbol{q}$ that will

decide the final value of the operation. Finally, we can also build the OR operation based on similar principles as

$$\text{OR} := \lambda \boldsymbol{p}.\ \lambda \boldsymbol{q}.\ \boldsymbol{p}\mathcal{T}\boldsymbol{q} \tag{5.30}$$

We select true if $\boldsymbol{p}$ is true, if it is not true then perhaps $\boldsymbol{q}$ is true, so return $\boldsymbol{q}$. The value of $\boldsymbol{q}$ will decide the final result.

These Boolean operators and Church numerals now allow us to do conditional statements. For example, we can develop conditional statements such as one to check if a number is zero

$$\text{ISZERO} := \lambda \boldsymbol{n}.\ \lambda \boldsymbol{f}.\ \lambda \boldsymbol{x}.\ \boldsymbol{n}(\lambda \boldsymbol{x}.\ \mathcal{F})\mathcal{T} \tag{5.31}$$

Remember that a Church numeral takes input a function and an argument and the number it defines is the number of times the function is applied to the argument, The expression $\lambda \boldsymbol{x}.\ \mathcal{F}$ is a function that will always return false, so that the ISZERO operation above is only true when the input Church numeral does not apply this function to the argument and that can only happen for the Church numeral for zero.

5.3 Combinators

At this point, notice how each λ expression is merely different combinations of the arguments $\boldsymbol{x}$ and $\boldsymbol{y}$ or whatever argument variables we have used. We combine the arguments to obtain the different expressions and operations. For example, the identity I occurs within the definition of $\mathcal{F}$ and MUL $\mathcal{B}$ occurs in other expressions such as the successor.

Just as for Boolean algebra and linear algebra, where we define a minimal set of operators that can replicate all of the operations necessary in that system, we can define a non-redundant or minimal set of combinators, i.e. λ expressions without free variables that combine arguments in different ways. In this way, we can use combinators as the atoms of λ-Calculus and problem can be solved or algorithms constructed like molecules from these atomic structures just as atoms from the periodic table in chemistry. This modular approach was pioneered by Haskell Curry [Curry and Feys, 1958] and he provided distinct names for these expressions and reformulated λ-Calculus based on them.

Let us begin by constructing Boolean algebra in this way. We can rewrite Boolean λ expressions using combinators by defining the Kestrel combinator

$$\mathcal{T} := \mathcal{K} := \lambda \boldsymbol{x}.\ \lambda \boldsymbol{y}.\ \boldsymbol{x} \tag{5.32}$$

then the expression for false becomes

$$\mathcal{F} := \mathcal{K}I \tag{5.33}$$

because we know that $\mathcal{K}I\boldsymbol{x} = I$, but I is a function that takes an argument, so we can write $\mathcal{K}I\boldsymbol{x}\boldsymbol{y} = I\boldsymbol{y} = \boldsymbol{y}$, which is equivalent to $\mathcal{F}$. Sometimes the $\mathcal{K}$ combinator is considered a constant or 'const' operator (such as in a language like Haskell), because it will always return the first argument regardless of the second or even whether $\mathcal{K}$ is defined without the second argument.

We can likewise define the Cardinal combinator and redefine the NOT function as

$$\text{NOT} := \mathcal{C} := \lambda \boldsymbol{f}.\ \lambda \boldsymbol{x}.\ \lambda \boldsymbol{y}.\ \boldsymbol{fyx} \tag{5.34}$$

that flips its arguments before applying the function $\boldsymbol{f}$, so that we get $\mathcal{CK} = \mathcal{KI}$ etc. We can also refine the OR operator to use the self application combinator called the Mockingbird combinator

$$\text{OR} := \mathcal{M} := \lambda \boldsymbol{f}.\ \boldsymbol{ff} \tag{5.35}$$

by noting that OR can be written from equation (5.30) as $\lambda \boldsymbol{p}.\ \lambda \boldsymbol{q}.\ \boldsymbol{ppq}$, since if $\boldsymbol{p}$ is true then we have completed the evaluation. For arithmetic, we can do the same for MUL and in fact we have already by defining MUL as the Bluebird combinator $\mathcal{B}$ in equation (5.21).

Haskell Curry [Curry and Feys, 1958] and others showed that all of λ-Calculus can be created by just using the combinators $\mathcal{B}$, $\mathcal{C}$, $\mathcal{K}$ and I. Although the details of this formulation is out of scope of this book, we will use some of these results to create one of the most famous results in computer science, the $\mathcal{Y}$ combinator.

5.4 The Y-Combinator

In previous sections, we showed that we can compute conditional branches (via Boolean algebra) and arithmetic via Church numerals. However, we need to augment at least one more operation to λ-Calculus to make it Turing complete: the ability to infinitely loop. One way to obtain infinite loops is to do computations recursively until a stopping condition is reached. The surprising result is that in a system where only unary, pure functions exist, we can still create the notion of recursion using the Mockingbird combinator as a starting point.

Notice that for $\mathcal{M}$, we apply the input function $\boldsymbol{f}$ to itself. When we apply $\mathcal{M}$ to itself, we get an expression that repeatedly calls itself ***ad infinitum***, so that we get an infinite recursion but nothing that is useful because it has no stopping condition. Interestingly, the self application of the Mockingbird to itself is related to the halting problem we discussed in a previous chapter. In general, we cannot know if any λ expression will always have a β reduction and halt or continue forever.

What we need is a recursive combinator that takes in a function $\boldsymbol{f}$, but calls $\boldsymbol{f}$ again with the recursive combinator only if the base condition is not satisfied. If the base condition is satisfied, we would like to stop calling it itself and allow β reduction. This is exactly what the $\mathcal{Y}$ combinator allows us to do and is defined as

$$\begin{aligned} \mathcal{Y} &:= \lambda \boldsymbol{f}.\ (\lambda \boldsymbol{x}.\ \boldsymbol{f}(\boldsymbol{xx}))(\lambda \boldsymbol{x}.\ \boldsymbol{f}(\boldsymbol{xx})) && (5.36)\\ &:= \lambda \boldsymbol{f}.\ \mathcal{M}(\lambda \boldsymbol{x}.\ \boldsymbol{f}(\mathcal{M}\boldsymbol{x})) && (5.37) \end{aligned}$$

We can see how this might work from a top level β reduction involving some combinator $\mathcal{R}$ as

$$\begin{aligned} \mathcal{YR} &= (\lambda \boldsymbol{f}.\ (\lambda \boldsymbol{x}.\ \boldsymbol{f}(\boldsymbol{xx}))(\lambda \boldsymbol{x}.\ \boldsymbol{f}(\boldsymbol{xx})))\mathcal{R} && (5.38)\\ &= (\lambda \boldsymbol{x}.\ \mathcal{R}(\boldsymbol{xx}))(\lambda \boldsymbol{x}.\ \mathcal{R}(\boldsymbol{xx}))\\ &= \mathcal{R}((\lambda \boldsymbol{x}.\ \mathcal{R}(\boldsymbol{xx}))(\lambda \boldsymbol{x}.\ \mathcal{R}(\boldsymbol{xx})))\\ &= \mathcal{R}(\mathcal{YR}) && (5.39) \end{aligned}$$

To demonstrate this amazing structure in a practical sense, we will show how to build the factorial function $\boldsymbol{n}!$ for the number $\boldsymbol{n}$. Consider the recursive form of the factorial function

$$\boldsymbol{n}! = \begin{cases} 1, & \text{if } \boldsymbol{n} = 0 \\ \boldsymbol{n}(\boldsymbol{n}-1)!, & \text{for } \boldsymbol{n} > 0. \end{cases} \tag{5.40}$$

The usual recursive form of the factorial function in most procedural languages, such as Python or C/C++ would be

```
1  int factorial(int number)
2  {
3     if (number == 0)
4        return 1;
5     else
6         return number * factorial(number − 1);
7  }
```

We can attempt to convert the above procedural 'function' directly to λ-Calculus for which we might (incorrectly) write

$$\text{FAC} = \lambda\boldsymbol{n}.\ \text{if ISZERO}(\boldsymbol{n})(1)\ \text{MUL}(\boldsymbol{n})(\text{FAC}(\text{PRED}(\boldsymbol{n}))) \tag{5.41}$$

Instead, because functions in λ-Calculus are anonymous, we can write the above correctly using the $\mathcal{Y}$ combinator while setting $\mathcal{Z}$ for ISZERO as

$$\begin{aligned} \text{FAC} &= \mathcal{Y}(\lambda\boldsymbol{f}.\ \lambda\boldsymbol{n}.\ \mathcal{Z}(\boldsymbol{n})(1)\ \text{MUL}(\boldsymbol{n})(\boldsymbol{f}(\text{PRED}(\boldsymbol{n})))) && (5.42)\\ &= \mathcal{Y}\mathcal{R} && (5.43) \end{aligned}$$

and set $\mathcal{R} = \lambda\boldsymbol{f}.\ \lambda\boldsymbol{n}.\ \mathcal{Z}(\boldsymbol{n})(1)\ \text{MUL}(\boldsymbol{n})(\boldsymbol{f}(\text{PRED}(\boldsymbol{n})))$. For example, we can compute $1!$ using the above expression as

$$\begin{aligned} \text{FAC 1} &= \mathcal{Y}(\mathcal{R})\ 1 && (5.44)\\ &= \mathcal{R}(\mathcal{Y}(\mathcal{R}))\ 1 && (5.45) \end{aligned}$$

Expanding the λ expressions

$$\begin{aligned} \text{FAC 1} &= \lambda\boldsymbol{f}.\ \lambda\boldsymbol{n}.\ \text{Z}(\boldsymbol{n})(1)\ \text{MUL}(\boldsymbol{n})(\boldsymbol{f}(\text{PRED}(\boldsymbol{n})))(\mathcal{Y}(\lambda\boldsymbol{f}.\ \lambda\boldsymbol{n}.\ \text{Z}(\boldsymbol{n})(1)\ \text{MUL}(\boldsymbol{n})(\boldsymbol{f}(\text{PRED}(\boldsymbol{n}))))\ 1\\ &= \lambda\boldsymbol{n}.\ \text{Z}(\boldsymbol{n})(1)\ \text{MUL}(\boldsymbol{n})((\mathcal{Y}(\lambda\boldsymbol{f}.\ \lambda\boldsymbol{n}.\ \text{Z}(\boldsymbol{n})(1)\ \text{MUL}(\boldsymbol{n})(\boldsymbol{f}(\text{PRED}(\boldsymbol{n})))))(\text{PRED}(\boldsymbol{n})))\ 1 \end{aligned}$$

Now we evaluate the 1 as the argument $\lambda\boldsymbol{n}$ and reduce the $\mathcal{Z}$, since 1 is not zero and so compute the false part of the statement

$$\begin{aligned} \text{FAC 1} &= \text{MUL}(1)(\mathcal{Y}(\lambda\boldsymbol{f}.\ \lambda\boldsymbol{n}.\ \text{Z}(\boldsymbol{n})(1)\ \text{MUL}(\boldsymbol{n})(\boldsymbol{f}(\text{PRED}(\boldsymbol{n})))))(\text{PRED}(1))\\ &= \text{MUL}(1)(\mathcal{R}(\mathcal{Y}\mathcal{R}))(\text{PRED}(1))\\ &= \text{MUL}(1)(\mathcal{R}(\mathcal{Y}\mathcal{R}))\ 0 && (5.46) \end{aligned}$$

Notice how the expression remaining is similar to equation (5.45) except requiring the evaluation with zero. The zero will trigger the base case and return a 1, allowing the recursion to reduce to the final result.

The $\mathcal{Y}$ combinator shows that even a simple system based on a compact notation system can encode not only recursion, but in conjunction with other combinators, computation itself can be reduced to just a handful of fixed modular combinations of functions and their arguments.

Abbreviations

1D one dimensional 19
2D two dimensional 11
3D three dimensional iii
GCD greatest common divisor 8
ACE automatic computing engine 18
FSM Finite State Machine iv

Bibliography

Berlekamp, E., J.Conway, R.Guy, 1982. Winning Ways for your Mathematical Plays. Vol. 2.

Church, A., 1932. A set of postulates for the foundation of logic. Annals of Mathematics 33 (2), 346–366.
URL https://doi.org/10.2307/1968337

Curry, H. B., Feys, R., 1958. Combinatory Logic, Volume I. North-Holland.

Diophantus, 100. Arithmetica. Springer-Verlag Berlin and Heidelberg GmbH & Co. KG (December 31, 1982).

Euclid, 300BCE. The Elements.

Euler, L., 1763. Theoremata Arithmetica Nova Methodo Demonstrata. Novi Commentarii Academiae Scientiarum Petropolitanae 8, 74–104.

Feynman, R., 2005. The Pleasure of Finding Things Out: The Best Short Works of Richard P. Feynman. Helix Books.
URL https://www.amazon.com/Pleasure-Finding-Things-Out-Richard-ebook/dp/B005QBLHB4

Gardner, M., 1970. Mathematical games: The fantastic combinations of john conway□s new solitaire game □life□. Scientific American (223), 120–123.

Gauss, C. F., 1801. Disquisitiones Arithmeticae. Yale Univeristy Press.

Gödel, K., Dec 1931. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. Monatshefte für Mathematik und Physik 38 (1), 173–198.
URL https://doi.org/10.1007/BF01700692

Klein, F., Mar 1893. Vergleichende betrachtungen über neuere geometrische forschungen. Mathematische Annalen 43 (1), 63–100.
URL https://doi.org/10.1007/BF01446615

Langton, C. G., 1986. Studying artificial life with cellular automata. Physica D: Nonlinear Phenomena 22 (1), 120 – 149, proceedings of the Fifth Annual International Conference.
URL https://doi.org/10.1016/0167-2789(86)90237-X

Sipser, M., 2013. Introduction to the theory of computation / Michael Sipser., 3rd Edition. Cengage Learning, Andover.

Turing, A. M., 1937. On computable numbers, with an application to the entscheidungsproblem. Proceedings of the London Mathematical Society s2-42 (1), 230–265.
URL https://doi.org/10.1112/plms/s2-42.1.230