Recursion and the λ-calculus

Miles Shang

LING 481: Joint Honours Thesis
McGill University

Supervised by:
Prof. Brenden Gillon
Department of Linguistics
McGill University

April 14, 2012

Abstract

The present paper investigates the relationship between the general recursive functions and the λ-definable functions. I discuss three proofs relating to these classes of functions. Two of these proofs employ Gödel numberings, and thus provide a good introduction to this technique. No background in metamathematics, computability, or recursion theory is assumed.

Contents

1 Introduction

It is a commonly cited fact that there exist several systems lying at the intersection of mathematics and computer science that are equivalent in some sense. The usual suspects are the "recursive" functions, the λ-calculus, and the Turing machine. Each of these systems comprehends a vast theory of its own. Therefore, it is important to ask exactly what is meant when we say that these systems are equivalent to one another. The present paper aims to answer this question for the particular cases of the recursive functions and the λ-calculus. The equivalence which is commonly referred to is in fact an equivalence between classes of functions from the n-tuples of natural numbers to the natural numbers. In other words, functions $f : \mathbb{N}^n \to \mathbb{N}$. In the rest of this paper, I will undertake to define the classes in question and to give proofs for their equivalence. In this endeavour, I aim to make this material understandable for readers who may not have an extensive background in formal methods.

My plan of attack is the following: In section 2, I will examine the history of the previously mentioned systems. In section 3, I will give a few definitions of "recursive function" and I will reproduce a proof that two of these definitions, which are of different natures, are in fact equivalent. In section 4, I will investigate the λ-calculus, honing in on the aspect of the theory that is relevent for the present paper. In section 5, I will reproduce the proofs that the classes defined in sections 3 and 4 are equivalent. Finally, in section 6, I will discuss the Church-Turing thesis, which was a driving motivation for the study of the above systems.

2 History

The history of "recursive function theory," comprising both of the systems which I investigate in the present paper, is given a thorough treatment in [15]. The author of that testimony, Stephen C. Kleene, was particularily well-placed to give an account of developments in this area as he was himself, as we shall see, a notable contributor. Another notable contributor, Barkley Rosser, gave an account of the history of the λ-calculus in [18]. Both Kleene and Rosser were students of Alonzo Church, inventor of the λ-calculus. A more recent resource, which I shall also rely on, is a paper with a greater focus on the notion of computability [19].

Computability refers to an intuitive notion: that of whether or not, for a given well-defined function or problem, a set of rules can be given that will allow a computer to calculate the function or solve the problem. A computer need not refer to what we think of as a computer, but could be a machine or even a human executing the given rules "in a desultory manner," as Alan Turing put it [20]. The question of what problems can be solved by computer has its roots in a problem posed by David Hilbert, called the *Entscheidungsproblem*. This problem asks to find, given a formal theory for mathematics, an algorithm that can decide whether or not any given statement is true or false. Such an algorithm would theoretically reduce mathematics to a mechanical process akin to arithmetic.

The first successful attempt at this problem was mounted by Alonzo Church using his invention, the λ-calculus. The earliest form of the λ-calculus first appeared in a 1932 paper of Church [4]. However, this paper did not mention computability at all.

Instead, it dealt with propositional functions and presented the λ-calculus as a system for the foundation of logic. As such, it introduced more notation than is necessary for studying computability, such as the logical connectives. In 1935, Kleene and Rosser discovered a paradox in this system, [16]. However, in his landmark paper of 1936 [5], Church repurposed his system into what is now known as the untyped λ-calculus, which is the system that is of interest to us in the present work. He used this system to prove that the *Entscheidungsproblem* of Russell and Whitehead's *Principia Mathematica* is unsolvable. As it turns out, Alan Turing independent proved the same result using his own invention, what is now known as the Turing machine [20]. Although Church's paper beat Turing's paper to press by a few months, Turing's paper is at least as famous as Church's, if not more so, as his techniques laid the foundations of modern computer science.

At around the same time, there was some interest in investigating computability in terms of recursion. According to Kleene, the first definition of the "general recursive functions" was developed by Kurt Gödel from a suggestion by Jacques Herbrand. The earliest known reference is a 1934 lecture by Gödel at the Institute of Advanced Study [10]. This work can be seen as a follow-up to Gödel's 1931 paper in which he proves his famous incompleteness theorems [11]. In that celebrated paper, Gödel made important use of the notion of primitive recursiveness. However, this notion, while sufficient for the purposes of that paper, was insufficient for the study of computability. At the time, it had been known for several years that there exist certain functions that are not definable under primitive recursion even though they meet intuitive criteria for computability. The Herbrand-Gödel definition was designed to include these functions. Although inspired by computability theory, this definition was entirely metamathematical in nature. As a result, it was not trivial to relate the Herbrand-Gödel definition of general recursion to more mechanical definitions of computability. In 1936, such a link was provided by Kleene's Normal Form Theorem [12].

3 Recursive functions

Two classes of functions are of interest to us: the *primitive recursive functions* and the *general recursive functions*. These appellations and definitions are taken from a textbook by Kleene [14]. We shall see that the primitive recursive functions are a strict subset of the general recursive functions. It is only this latter class which are generally identified with "computable" or "recursive" functions.

3.1 Primitive recursive functions

To begin with, we define our building blocks. Kleene calls these the *initial functions*. All of these should be considered as total functions.

(I) The *successor function*.
$\varphi(x) = S(x) = x + 1$

(II) The *constant functions*.
$\varphi(x_1, ..., x_n) = q$, a constant

(III) The *identity functions*, sometimes called the *projection functions*.
$\varphi(x_1, ..., x_n) = x_i$ for some $1 \leq i \leq n$

Next, we need some mechanism to combine these simple functions to create more complicated ones. Both (IV) and (V) are schemata that take in functions $\psi, \chi.\chi_1, ..., \chi_m$ and produce a new function φ.

(IV) *Definition by substitution*, also known as *function composition*.
$\varphi(x_1, ..., x_n) = \psi(\chi_1(x_1, ..., x_n), ..., \chi_m(x_1, ..., x_n))$

(V) *Primitive recursion*.

(a) $\begin{cases} \varphi(0) = q, \text{ a constant} \\ \varphi(x+1) = \chi(x, \varphi(x)) \end{cases}$

(b) $\begin{cases} \varphi(0, x_2, ..., x_n) = \psi(x_2, ..., x_n) \\ \varphi(x_1 + 1, x_2, ..., x_n) = \chi(x_1, \varphi(x_1, x_2, ..., x_n), x_2, ..., x_n) \end{cases}$

Primitive recursion represents the same process as is seen in the definition of the natural numbers, called *definition by induction*. In generating the natural numbers, we start with an element called 0 and a successor function $S(x)$. The number 1 is defined as $S(0)$. The number 2 is defined as $S(1) = S(S(0))$, etc. Similarly, in primitive recursion, we set a value for $\varphi(0)$ and define all further values by repeated function application. This is also similar to proof by induction. We may think of $\varphi(0) = q$ as the base case and $\varphi(x+1) = \chi(x, \varphi(x))$ as the induction step.

Intuitively, any function that can be constructed from (I), (II), and (III) using as many applications of (IV) and (V) as necessary is called a *primitive recursive (p.r.) function*. To formalize this, we define a *primitive recursive description* as a finite, ordered sequence of functions $\varphi_1, ..., \varphi_n$ where each function in the sequence is either an initial function, or is obtained from preceding functions by an application of (IV) or (V). This is analogous to derivations in first-order logic, where the axioms are the initial functions, and the rules of inference are the schemata (IV) and (V). If $\varphi_1, ..., \varphi_n$ is a sequence which satisfies these criteria, then it is said to describe φ_n. Any function for which such a description can be given is called primitive recursive.

Example 1. (from [14]) Consider the function $f(x, y) = x + y$. It can be described as follows:

$f_1(x) = x + 1$ (successor function)

$f_2(x) = x$ (identity function)

$f_3(x, y, z) = y$ (identity function)

$f_4(x, y, z) = f_1(f_3(x, y, z)) = y + 1$ (IV) applied to f_1, f_3

$\begin{cases} f(0, y) = f_2(y) = y \\ f(x+1, y) = f_4(x, f(x, y), y) = f(x, y) + 1 \end{cases}$ (Vb) applied to f_2, f_4

This shows that f is primitive recursive.

Other examples of p.r. functions include multiplication, exponentiation, factorial, min, max, and many others. Sketches of the descriptions are given in Kleene's textbook [14].

3.2 General recursive functions

It seems that many functions that we can think of are primitive recursive. Does this class encompass our intuitive understanding of "computable" functions? In fact, it cannot account for at least one example, known as the Ackermann function. If we think of multiplication as repeated addition, and exponentiation as repeated multiplication, we can imagine repeated exponentiation, repeated repeated exponentiation, etc. Ackermann showed that a function which parametrizes this hierarchy of operations grows too quickly to be p.r. Here is one version of the Ackermann function ξ, from [14]:

$$\alpha(n,a) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ a & \text{otherwise} \end{cases} \tag{1}$$

$$\begin{cases} \xi(0,b,a) & = a+b \\ \xi(n+1,0,a) & = \alpha(n,a) \\ \xi(n+1,b+1,a) & = \xi(n,\xi(n+1,b,a),a) \end{cases} \tag{2}$$

The definition of ξ can be described as "double recursion" because it recurses on both n and b. It does not follow the schema of primitive recursion and in fact could not be reduced to primitive recursion. However, the definition of ξ is very nearly an algorithm to compute values of ξ for any given input. Therefore, it is clear that ξ is computable in an intuitive sense. In order to arrive at a definition of recursiveness that includes ξ and similar functions, we must abandon, for the moment, definitions that build up recursive functions from simple functions by means of schemata. Instead, we turn to a definition that seems to have nothing at all to do with recursion.

Although the definition of the Ackermann function ξ does not employ the schemata of primitive recursion, it resembles those schemata in that it consists of a system of equations. This is the intuition for the Herbrand-Gödel definition of recursive functions. In order to formalize this, the first step is to specify what systems of equations are admissible for this purpose. Then, we define rules of inference for passing from equation to equation, with the end goal of an equation that explicitly gives a value for a function applied to a specific input. For example, we would like to be able to obtain, from the equations (1) and (2) that $\xi(1,1,1) = 1$.

The following definitions are taken from [12]. However, we will use some additional conventions: We use a dot to mark numerals, and bold face to mark metavariables for expressions which are not necessarily individual symbols.

An *expression* is a finite sequence of the following symbols:

- $\dot{0}$ (the numeral zero)
- S (the successor function)
- $w_0, w_1, ...$ (numerical variables)
- $\varrho_0, \varrho_1, ...$ (variables for functions of $r_1, r_2, ...$ arguments)
- ( ) , = (parentheses, comma, equality sign)

A *term* is defined thus:

- $\dot{0}, w_0, w_1, ...$ are terms.

- If $\mathbf{a_1}, \mathbf{a_2}, ...$ are terms, then $S(\mathbf{a_1}), \varrho_0(\mathbf{a_1}, ..., \mathbf{a_{r_0}}), \varrho_1(\mathbf{a_1}, ..., \mathbf{a_{r_1}}), ...$ are terms.

By *numeral* is meant one of the expressions $\dot{0}, S(\dot{0}), S(S(\dot{0})), ...$ If $\mathbf{a}$ and $\mathbf{b}$ are terms (and if $\sigma_1, ..., \sigma_n$ are functional variables such that at least one of $\sigma_1, ..., \sigma_n$ occurs in $\mathbf{a}$ or $\mathbf{b}$, but no functional variables other than $\sigma_1, ..., \sigma_n$ occur in $\mathbf{a}$ or $\mathbf{b}$), $\mathbf{a} = \mathbf{b}$ will be called an *equation* (*in* $\sigma_1, ..., \sigma_n$). By a *system* of equations we mean a finite sequence of equations.

To the above, we may add the following remarks:

- Numerals, terms, and equations are all examples of expressions. However, systems of equations are not expressions.
- We will abbreviate $S(\dot{0})$ as $\dot{1}$, $S(S(\dot{0}))$ as $\dot{2}$, etc. Note, however, that $\dot{1}$ and 1 are not the same. The former refers to the numeral $S(\dot{0})$, considered as a sequence of symbols within our formal system. The latter *is* a numeral, outside of our formal system, which refers to a number.

Example 2. The equations (1) and (2) can be made into an admissible system of equations by expanding the definitions of α and $+$.

$$\begin{cases} \alpha_0(\dot{0}, a) & = \dot{1} \\ \alpha_0(S(n), a) & = a \\ \alpha(\dot{0}, a) & = \dot{0} \\ \alpha(S(n), a) & = \alpha_0(n, a) \\ \xi(\dot{0}, \dot{0}, a) & = a \\ \xi(\dot{0}, S(b), a) & = S(\xi(\dot{0}, b, a)) \\ \xi(S(n), \dot{0}, a) & = \alpha(n, a) \\ \xi(S(n), S(b), a) & = \xi(n, \xi(S(n), b, a), a) \end{cases} \tag{3}$$

To be fully compliant, we should also replace our variables with the mandated ones. For example, α_0, α, ξ should be $\varrho_0, \varrho_1, \varrho_2$ and n, b, a should be w_0, w_1, w_2.

In our formal system, there are two rules of inference:

R1: Replace all numerical variables in an expression by numerals, so long as the same variable is always replaced by the same numeral.

R2: If $\mathbf{A}, \mathbf{B}, \mathbf{C}$ are expressions, to pass from $\mathbf{A}$ and $\mathbf{B} = \mathbf{C}$ to the result of substituting $\mathbf{C}$ for a particular occurrence of $\mathbf{B}$ in $\mathbf{A}$.

Example 3. How can we get from the system of equations (3) to $\xi(\dot{1}, \dot{1}, \dot{1}) = \dot{1}$? Here it is laid out in derivation form:

1. $\xi(S(n), S(b), a) = \xi(n, \xi(S(n), b, a), a)$ — Given.
2. $\xi(\dot{1}, \dot{1}, \dot{1}) = \xi(\dot{0}, \xi(\dot{1}, \dot{0}, \dot{1}), \dot{1})$ — R1 applied to 1 with $n \to \dot{0}$, $b \to \dot{0}$, $a \to \dot{1}$.
3. $\xi(S(n), \dot{0}, a) = \alpha(n, a)$ — Given.
4. $\xi(\dot{1}, \dot{0}, \dot{1}) = \alpha(\dot{0}, \dot{1})$ — R1 applied to 3 with $n \to \dot{0}$, $a \to \dot{1}$.
5. $\alpha(\dot{0}, a) = \dot{0}$ — Given.

6. $\alpha(\dot{0}, \dot{1}) = \dot{0}$ R1 applied to 5 with $a \to \dot{1}$.
7. $\xi(\dot{1}, \dot{0}, \dot{1}) = \dot{0}$ R2 applied to 4, 6.
8. $\xi(\dot{1}, \dot{1}, \dot{1}) = \xi(\dot{0}, \dot{0}, \dot{1})$ R2 applied to 2, 7.
9. $\xi(\dot{0}, \dot{0}, a) = a$ Given.
10. $\xi(\dot{0}, \dot{0}, \dot{1}) = \dot{1}$ R1 applied to 9 with $a \to \dot{1}$
11. $\xi(\dot{1}, \dot{1}, \dot{1}) = \dot{1}$ as required R2 applied to 8, 10.

Let $\mathbf{E}$ be a system of equations in $\sigma_1, ..., \sigma_n$. Suppose that for each i $(1 \leq i \leq n)$ and for each set of numerals $\dot{\mathbf{k}}_{\mathbf{1}}, ..., \dot{\mathbf{k}}_{\mathbf{s_i}}$, there exists exactly one numeral $\dot{\mathbf{k}}$ such that the equation $\sigma_i(\dot{\mathbf{k}}_{\mathbf{1}}, ..., \dot{\mathbf{k}}_{\mathbf{s_i}}) = \dot{\mathbf{k}}$ can be obtained from $\mathbf{E}$ by applying R1 and R2. Then we say that $\mathbf{E}$ *recursively defines* $\sigma_1, ..., \sigma_n$. If such an $\mathbf{E}$ exists, then σ_n is said to be *general recursive*.

Note that there is no mention, in this definition, of any schemata such as substitution and primitive recursion. I will leave it to the reader to convince himself that the primitive recursive functions are also general recursive under the Herbrand-Gödel definition.

The above definition of recursion is an unwieldy one, as for any non-trivial system of equations, it requires quite a bit of proof to show that the system indeed recursively defines its function variables. In [12], Kleene showed that the decision problem of whether or not an arbitrary system of equations recursively defines its function variables is not itself recursive. Gödel himself believed that the above definition adequately covers the intuitive notion of computability. However, because Gödel's definition was so different from the other conceptions at the time, Kleene would have to recast it into the language of schemata in order to prove anything useful about it.

To my knowledge, no modern source treats the Herbrand-Gödel definition as anything other than a historical note. For any practical purposes, the definition that will be given in the next subsection is vastly easier to work with. However, in my opinion, it is not as well-motivated as the Herbrand-Gödel definition.

3.3 μ-recursive functions

Let P be an predicate on the natural numbers. By this, we mean that P takes as input a natural number and returns true or false. Suppose that P does not uniformly return false (i.e. there is some x such that $P(x)$ is true). Then define $\mu x P(x)$ to be the least number x such that $P(x)$ is true. This is called the *μ-operator*, or *minimisation operator*. The term "operator" refers to the fact that μ takes predicates as input. If P is a 1-place predicate, then $\mu x P(x)$ is a number. If P is an n-place predicate with $n > 1$, then $\mu y P(x_1, ..., x_n, y)$ is a function in $x_1, ..., x_n$, and the following is a schema:

(VIa) $\varphi(x_1, ..., x_n) = \mu y P(x_1, ..., x_n, y)$

Now suppose $\chi(x_1, ..., x_n, y)$ is a function such that for all natural numbers $x_1, ..., x_n$, there exists some natural number y such that $\chi(x_1, ..., x_n, y) = 0$. Then we have the following schema:

(VIb) $\varphi(x_1, ..., x_n) = \mu y[\chi(x_1, ..., x_n, y) = 0]$

The goal of the present subsection is to show that any function φ that is recursive in the Herbrand-Gödel sense can be written using only a primitive recursive function V, a p.r. predicate E, and *one* application of the μ-operator: $\varphi(x_1, ..., x_r) = V(\mu y E(x_1, ..., x_r, y))$. This is the Kleene Normal Form Theorem. The proof that I shall present is from [12].

The first step is to set up a system by which we can assign a unique natural number to every possible term, expression, finite sequence of expressions, etc. This technique is called *Gödel numbering* (GN). It was first used by Gödel to prove his incompleteness theorems [11].

Recall that the symbols $w_0, w_1, ...$ are the numerical variables and $\varrho_0, \varrho_1, ...$ are the function variables. We associate symbols to numbers as follows:

Symbol	$\dot{0}$	S	$=$	,	(	)	w_i	ϱ_i
Gödel number	1	3	5	7	11	13	p_{i+7}	p_{i+7}^2

where p_i is the i^{th} prime number. Now if $N_1, ..., N_k$ are assigned numbers $n_1, ..., n_k$, respectively, then the sequence $N_1, ..., N_k$ is assigned the number $p_1^{n_1} \cdots p_k^{n_k}$. Note that we have not specified what $N_1, ..., N_k$ are. If the N_i are symbols, then the sequence $N_1, ..., N_k$ is an expression. If the N_i are equations, then the sequence $N_1, ..., N_k$ is a system of equations. One could also encode sequences of sequences of sequences, etc. I will use the term "item" to refer to any symbol, expression, sequence of expressions, etc. If an item is not a symbol and it is composed of $N_1, ..., N_k$, then I will refer to the N_i as the "components" of the item.

The process of Gödel numbering is fully reversible. Given a natural number n, by the Fundamental Theorem of Arithmetic, n can be reduced to its prime factors $n = p_1^{n_1} \cdots p_k^{n_k}$. Each of $n_1, ..., n_k$ can also be factorized. If each has only one prime factor, then the original n encoded an expression, and we can decode directly from $n_1, ..., n_k$ to symbols. Otherwise, we repeat the whole process for each of $n_1, ..., n_k$. Note that in our chosen numbering, the numbers assigned to symbols need not be primes, but they all have only one prime factor, i.e. they are all powers of primes.

Next, we expand the rules of inference R1 and R2 so that we can encode them as functions. After all, these rules can be thought of as taking in expressions and returning different expressions. Now that we have a way to encode expressions as natural numbers, we should be able to encode the action of R1 and R2 as functions on natural numbers. However, R1 and R2 each do not represent a single function, but a class of them, indexed in a certain way:

- R_{3i} : Replace all occurrences of w_i (the i^{th} numerical variable) in an expression with $S(w_i)$.
- R_{3i+1} : Replace all occurrences of w_i in an expression with the numeral $\dot{0}$.
- R_{3i+2} : To pass from $\mathbf{A}$ and $\mathbf{B} = \mathbf{C}$ to the result of replacing the occurrence of $\mathbf{B}$ in $\mathbf{A}$ beginning with the $i + 1^{\text{st}}$ symbol by $\mathbf{C}$, if there is such an occurrence.

From a given system of equations $\mathbf{E}$, the set of equations that can be obtained using R1 and R2 is the same as the set that can be obtained by using $R_{0,1,2,...}$. One single application of R1 can be broken up into a chain of applications of R_{3i} and R_{3i+1}. An application of R2 can be broken up into applications of R_{3i+2}.

Now the idea is to construct a sequence of primitive recursive functions, culminating in the functions into which we will decompose the given general recursive function. Some of the functions that will appear in our list are predicates on natural numbers. This is not a problem, as we will treat them in the natural way as functions from $\mathbb{N}^n$ to $\{0, 1\}$. Therefore, we may consider them simply as functions of natural numbers with a restricted range. As such, they may be labelled primitive recursive or not, according to our previous definition. At the same time, we may use the words "true" and "false" to stand in for 1 and 0, respectively.

In order to show that the functions in our list are p.r., we need a few theorems from [11] which I will state here without proof:

Theorem 1. *If A and B are p.r. predicates, then so are $\neg A$ (not A), $A \vee B$ (A or B), and $A \wedge B$ (A and B).*

Theorem 2. *If the functions $\phi(x_1, ..., x_n)$ and $\psi(y_1, ..., y_n)$ are p.r., then so is the predicate which asks, for inputs $x_1, ..., x_n$ and $y_1, ..., y_n$, whether or not $\phi(x_1, ..., x_n)$ is equal to $\psi(y_1, ..., y_n)$.*

Theorem 3. *If the function $\phi(x_1, ..., x_n)$ and the predicate $P(x, y_1, ..., y_n)$ are p.r., then so are the following predicates and functions:*

1. $S(x_1, ..., x_n, y_1, ..., y_n) = \exists x[[x \leq \phi(x_1, ..., x_n)] \wedge P(x, y_1, ..., y_n)]$
2. $T(x_1, ..., x_n, y_1, ..., y_n) = \forall x[[x \leq \phi(x_1, ..., x_n)] \rightarrow P(x, y_1, ..., y_n)]$
3. $\psi(x_1, ..., x_n, y_1, ..., y_n) = \varepsilon x[[x \leq \phi(x_1, ..., x_n)] \wedge P(x, y_1, ..., y_n)]$ *where $\varepsilon x P(x)$ means the least number x for which $P(x)$ is true and 0 if there is no such number.*

The following functions and predicates are all primitive recursive, which can be shown by using the above theorems. I omit a few intermediate functions that Kleene and Gödel used to prove this. In order to fill in the holes, the interested reader is directed to [11] and [12]. I will use the abbreviation GN to refer to the Gödel number corresponding to an item, and GN^{-1} to refer to the item corresponding to a number. Although this introduces verbosity, it allows us to maintain the convention that n, x, y, etc. shall be interpreted as natural numbers only, and not the items to which they correspond.

- Addition $(x + y)$, multiplication (xy), and exponentiation (x^y or $exp(x, y)$) of natural numbers, considered as functions of x and y.
- Subtraction $x - y$ defined as follows, in order to prevent a negative result:
  $x - y = \varepsilon z[[z \leq x] \wedge [x = y + z]]$
- $Div(x, y)$, integer division (or, more accurately, natural number division), defined as follows:
  $Div(x, y) = \varepsilon z[[z \leq x] \wedge [(z + 1)y > x]]$
- $Rem(x, y)$, which returns the remainder from dividing x by y.
- $Pr(n)$, which returns the n^{th} prime number, with the convention that $Pr(0) = 0$, followed by the usual $2, 3, 5$, etc.
- $l(x)$, which returns the number of components of the GN^{-1} of x, or equivalently, the number of prime factors of x.

- $Comp(k, x)$, which returns the GN of the k^{th} component of the GN^{-1} of x. Equivalently, if x has prime factorization $p_1^{m_1} \cdots p_k^{m_k} \cdots p_n^{m_n}$, then $Comp(k, x)$ returns m_k.
- $x \star y$, the GN of the concatenation of the GN^{-1}s of x and y. Note that if x and y are GNs of expressions, then $x \star y$ is the GN of an expression, not of a sequence of expressions.
- $\langle x \rangle = 2^x$, the GN of the item whose only component is the GN^{-1} of x. In other words, this embeds the GN of x one level deeper in the hierarchy of expressions, sequences, sequences of sequences, etc. It is used to obtain the GN of an expression consisting of only one symbol.
- $Dy(k) = 2^{a(k)}3^{b(k)}$ where $a(k)$ and $b(k)$ follow this pattern:

k	0	1	2	3	4	5	6	7	8	$\cdots$
$a(k)$	0	0	1	1	0	1	2	2	2	$\cdots$
$b(k)$	0	1	1	0	2	2	2	1	0	$\cdots$

  Dy encodes a particular way of enumerating pairs of natural numbers. In particular, $Dy(0)$ through $Dy((k)^2 - 1)$ encode all pairs with both components less than k. These pairs are encoded such that $Comp(1, Dy(k))$ returns the first number in the pair, and $Comp(2, Dy(k))$ returns the second, since 2 and 3 are the first and second primes.
- $Rule(n, x, y)$, which returns the GN of the result of applying rule R_n to the GN^{-1} of x. If this corresponds to a rule R_{3i} or R_{3i+1}, then y is not used. In the case of a rule R_{3i+2}, the y is the GN of the equation $B = C$ from the definition of the rules R_{3i+2}. I omit here several steps of the proof necessary to construct the function $Rule$. I hope that such a construction is somewhat evident from the definitions of the R_i.
- $Z(n)$, which returns the GN of the numeral corresponding to n. For example, $Z(2)$ returns the GN of $S(S(\dot{0}))$.
- $Eval_p(n, y, x_1, ..., x_p) = (\exists x)[[x \leq y] \wedge$
  $[y = \langle Pr(n+7)^2 \rangle \star \langle 11 \rangle \star Z(x_1) \star \langle 7 \rangle \star \cdots \star \langle 7 \rangle \star Z(x_p) \star \langle 13 \rangle \star \langle 5 \rangle \star Z(x)]]$
  gloss: $\varrho_n \quad (\quad \dot{\mathbf{x}}_\mathbf{1} \quad , \quad , \quad \dot{\mathbf{x}}_\mathbf{p} \quad) \quad = \quad \mathbf{x}$
  The gloss should be read with the understanding that bold symbols represent corresponding numerals. Intuitively, $Eval$ asks whether or not some x exists such that y is the GN of the expression $\varrho_n(\dot{\mathbf{x}}_\mathbf{1}, ..., \dot{\mathbf{x}}_\mathbf{p}) = \dot{\mathbf{x}}$.
- $Val(y)$, which returns x if y is the GN of an expression of the form $\mathbf{a} = \dot{\mathbf{x}}$, where $\mathbf{a}$ is any expression.

Consider the function $Rule'(n, x, y)$, defined as follows:

$$Rule'(0, x, y) = x$$
$$Rule'(n + 1, x, y) = Rule(n, x, y)$$

Since $Rule(n, x, y)$ corresponds to applying the rule R_n, therefore $Rule'(n, x, y)$ corresponds to applying the rule R_{n-1} for $n > 0$, and doing nothing (applying no rule at all) for $n = 0$. Now consider the functions $L(k, z)$, and $T(k, z)$, defined as follows:

$$L(0,z) = l(z)$$

$$L(k+1,z) = (k+1)L(k,z)^2$$

$$T(0,z) = z$$

$$T(k+1,z) = \prod_{n=0}^{L(k+1,z)-1} \begin{array}{l} exp(Pr(n+1), (Rule'(Div(n, L(k,z)^2), \\ Comp(Comp(1, Dy(Rem(n, L(k,z)^2))) + 1, T(k,z)), \\ Comp(Comp(2, Dy(Rem(n, L(k,z)^2))) + 1, T(k,z))))) \end{array}$$

Although our goal is to construct a pair of primitive recursive functions for each specific system of equations **E** from the Gödel-Herbrand definition, the exact choice of **E** has not yet come into play. That is to say, the definitions of the above functions are all independent of **E**. Now we finally use **E**.

Let e, and hence $T(0,e)$, be the GN of **E**. By definition, **E** has $l(e) = L(0,e)$ components. For all k, define S_k to be the GN^{-1} of $T(k,e)$. In particular, S_0 is **E**. We can see that roughly speaking, $T(k+1,e)$, the GN of S_{k+1}, has the form $\prod_{n=0}^{L(k+1,e)-1}(Pr(n+1))^{X_{n,k}}$. If we omit the subscript k, and we say that the X_n are the GNs of $\mathbf{X_n}$, then $T(k+1,e)$ is the GN of the sequence $\mathbf{X_1}, \mathbf{X_2}, ..., \mathbf{X_{L(k+1,e)}}$. This sequence has $L(k+1,e)$ components. Therefore, S_k has $L(k,e)$ components for all k.

Examining the definition of T in more detail, we recognize that the form of the sequence X_n can be further analyzed as $Rule'(A_n, B_n, C_n)$. The components of S_{k+1} are therefore obtained in some way from the components of S_k by applying a rule R_{A_n}. In fact, B_n and the C_n will take as values the GNs of the components of S_k, and A_n will take the values $0, ..., k$. There are $L(k+1,e) = (k+1)L(k,e)^2$ possible combinations of these values and all of them will be exhausted as n goes from 0 to $L(k+1,e) - 1$. Therefore, an item is a component of S_{k+1} if and only if it is a component of S_k or it can be obtained from any combination of components of S_k using any of the rules R_0 through R_k. As k varies over the natural numbers, all equations derivable from **E** will be components of some S_k. No other expressions will be components of any S_k. To extract all of these, we define the function $C(e,m) = Comp(a(m,e), T(b(m,e),e))$ where a and b take values as follows:

m	0	1	$\cdots$						
$a(m,e)$	0	0	$\cdots$	0	1	1	$\cdots$	1	$\cdots$
$b(m,e)$	$L(0,e)$	$L(0,e)-1$	$\cdots$	1	$L(1,e)$	$L(1,e)-1$	$\cdots$	1	$\cdots$

Now, as m runs through the natural numbers, $C(e,m)$ will run through the GNs of all expressions obtainable from **E** by applying rules R1 and R2. If a set of numbers is the range of a p.r. function, we say that that set is *recursively enumerable*. If we extend this definition to GNs, then since C is p.r., we can say that the set of expressions obtainable from **E** by applying rules R1 and R2 is recursively enumerable.

If **E** recursively defines φ, which has arity r, and φ is represented by the function symbol ϱ_a in **E**, then define $E(x_1, ..., x_r, y) = Eval_r(a, C(e,y), x_1, ..., x_r)$ and $V(y) = Val(C(e,y))$. The Herbrand-Gödel definition says that for any set of inputs $x_1, ..., x_r$, there exists a unique x such that the equation $\varrho(\dot{\mathbf{x}}_\mathbf{1}, ..., \dot{\mathbf{x}}_\mathbf{r}) = \dot{\mathbf{x}}$ is obtainable.

Equivalently, for all $x_1, ..., x_r$, there exists a y such that $E(x_1, ..., x_r)$ is true. Therefore, $\mu y E(x_1, ..., x_r, y)$ is well-defined and $\varphi(x_1, ..., x_r) = V(\mu y E(x_1, ..., x_r, y))$, as required.

More proof is required in order to show that the μ-operator is recursive according to the Herbrand-Gödel definition. However, once this is done, then we have a new, much more applicable definition of recursiveness. This definition, for what we call the *μ-recursive functions*, is identical to the definition of the primitive recursive functions but for the addition of the minimization schema (VIb).

4 The λ-calculus

The λ-calculus attempts to capture several aspects of the theory of functions that are not captured in the usual notation. For example, say we define a function $f(x) = x^2$. Firstly, the name which we have assigned, f, is arbitrary, and in some cases, f may as well be nameless. For this reason, we have the notation $x \mapsto x^2$. In addition, we may want our system to recognize the fact that $x \mapsto x^2$ and $y \mapsto y^2$ are essentially the same function. The name of the independent variable doesn't matter. Finally, we wish to capture the notions of free and bound variables. For example, in calculus, if we integrate $\int_0^1 x \sin(y) dy$, the $\int \cdots dy$ symbol binds y, but not x, so the result will be in terms of x, but not y. This distinction between free and bound has interesting consequences. As Kleene has pointed out [15], the expression $x^4 + 3x^2 + 2$ can be considered both as a function and as a number (a different one for each x). An even function f is one where for all x, $f(x) = f(-x)$. If we say that "$x^4 + 3x^2 + 2$ is even," we are making a statement about the function. However, if we say that "$x^4 + 3x^2 + 2$ is not less than two," this is a statement about the values of the function. This ambiguity is resolved by introducing a binding symbol λ, so that the expression $\lambda x(x^4 + 3x^2 + 2)$ refers to the function. In this expression, x is bound by λ, which unlike $\forall$ and $\exists$, does not quantify, but simply denotes that x is a placeholder.

In the previous examples, we have not specified what kinds of values, or *type* our variables can take. This may be important for certain statements that can be made about functions. The present section investigates an untyped version of the λ-calculus. Because the variables in such a system are not specified for what they can represent, it is best to think of them as representing nothing at all. They should be considered only as symbols, with no meaning outside of the system.

In this section, I will mainly work from Church's landmark paper [5]. Other good sources include [3], [2], and [1].

4.1 Definition

Here is an early definition of the λ-calculus, quoted directly from Church [5], with some changes in formatting:

> We select a particular list of symbols, consisting of the symbols {, }, (,), λ, [,], and an enumerably infinite set of symbols $a, b, c, ...$ to be called *variables*. And we define the word *formula* to mean any finite sequence of symbols out of this list. The terms *well-formed formula*, *free variable*, and *bound variable* are then defined by induction as follows.

- A variable $\mathbf{x}$ standing alone is a well-formed formula and the occurrence of $\mathbf{x}$ in it is an occurrence of $\mathbf{x}$ as a free variable in it.
- If the formulas $\mathbf{F}$ and $\mathbf{X}$ are well-formed, $\{\mathbf{F}\}(\mathbf{X})$ is well-formed and an occurrence of $\mathbf{x}$ as a free (bound) variable in $\mathbf{F}$ or $\mathbf{X}$ is an occurrence of $\mathbf{x}$ as a free (bound) variable in $\{\mathbf{F}\}(\mathbf{X})$.
- If the formula $\mathbf{M}$ is well-formed and contains an occurrence of $\mathbf{x}$ as a free variable in $\mathbf{M}$, then $\lambda\mathbf{x}[\mathbf{M}]$ is well-formed, and occurrence of $\mathbf{x}$ in $\lambda\mathbf{x}[\mathbf{M}]$ is an occurrence of $\mathbf{x}$ as a bound variable in $\lambda\mathbf{x}[\mathbf{M}]$, and an occurrence of a variable $\mathbf{y}$, other than $\mathbf{x}$, as a free (bound) variable in $\mathbf{M}$ is an occurrence of $\mathbf{y}$ as a free (bound) variable in $\lambda\mathbf{x}[\mathbf{M}]$.

$\{\mathbf{F}\}(\mathbf{X})$, sometimes called an *application*, can be thought of as an expression representing the function $\mathbf{F}$ with input $\mathbf{X}$. It can be abbreviated as $\mathbf{F}(\mathbf{X})$. For repeated application, for example $M(a)(b)$, we can write $M(a, b)$. $\lambda\mathbf{x}[\mathbf{M}]$, sometimes called an *abstraction*, can be thought of as *binding*, or selecting one variable in $\mathbf{M}$ to act as independent variable, the rest remaining as dummy variables. We can call $\mathbf{x}$ the *bound variable* of the abstraction and $\mathbf{M}$ the *scope*. The abstraction can be abbreviated as $\lambda\mathbf{x} \cdot \mathbf{M}$. Furthermore, in the case of several abstractions in a row, for example $\lambda a[\lambda b[a(b)]]$, we can write $\lambda ab \cdot a(b)$.

When it comes to labelling variables as free or as bound, we note two facts. Firstly, the labels "free" and "bound" are relative to the formula being considered. For example, we say that x occurs as a bound variable in $\lambda x \cdot x(x)$ and as a free variable in $x(x)$ even though the latter is a subformula of the former. Secondly, a variable can occur as both bound and free in the same pff, for example, x in $x(\lambda x \cdot x)$. However, a particular *occurrence* of a variable can not be both bound and free.

4.2 λ-I-calculus vs. λ-K-calculus

The above definition differs from most modern definitions of the untyped λ-calculus in only one substantive respect. In the definition of abstraction, $\lambda\mathbf{x}[\mathbf{M}]$, it is required that $\mathbf{x}$ occur in $\mathbf{M}$. This means that, for example, $\lambda x \cdot y$ is not a well-formed formula. In contrast, most modern definitions do not include this requirement, so that $\lambda x \cdot y$ is well-formed. I will follow the terminology of Kleene [13] in contrasting between properly formed formulas and well-formed formulas. To define a properly formed formula, replace every occurrence of "well-formed" in the above definition with "properly formed" and remove the requirement in item 3 that $\mathbf{M}$ contain an occurrence of $\mathbf{x}$ as a free variable in $\mathbf{M}$.

Note that under this new definition, for properly- but not well-formed formulas such as $\lambda x \cdot \lambda x \cdot x$, x is bound by the nearest abstraction. In this example, the first λ does not bind any occurrence of x, and the second λ binds the final (and technically, the only) occurrence of x.

Many sources refer to the original definition of Church as the λ-I-calculus, and the later definition as the λ-K-calculus [3,15,18]. Church himself was an adament proponent of the λ-I-calculus over the λ-K-calculus [3]. According to Kleene, the latter introduces "difficulties ... in the formal logics in which this theory is used" [13]. Because we are not currently interested in these formal logics, and because all modern sources use the

λ-K-calculus, we shall seek out the "simplification ... afforded in the proofs of many theorems" [13] by considering only the λ-K-calculus. Future references to the λ-calculus should be taken to mean the λ-K-calculus.

4.3 Church numerals

As defined thus far, the λ-calculus is a useful tool for reasoning about functions. However, it seems that in order to create a useful system, we would need to specify types for the variables. For example, since in the rest of the present paper we consider functions of natural numbers, we might stipulate that variables in the λ-calculus represent natural numbers. Then we might introduce atomic functions from outside of the λ-calculus, such as the initial functions used to define the primitive recursive functions. In fact, this can be avoided by defining the natural numbers within the system itself, thus avoiding the need for elements outside of the theory. This is known as a Church encoding. The numerals themselves are known as Church numerals. There are several, trivially different Church encodings. In his original paper [5], Church himself used the following encoding:

$$\begin{aligned}\dot{1} &= \lambda f x \cdot f(x)\\ \dot{2} &= \lambda f x \cdot f(f(x))\\ \dot{3} &= \lambda f x \cdot f(f(f(x)))\\ &\vdots\end{aligned}$$

As we can see, Church worked with $\mathbb{N}\backslash\{0\}$, the positive integers. Kleene [13] used the following encoding in one of his papers:

$$\begin{aligned}\dot{0} &= \lambda f x \cdot f(x)\\ \dot{1} &= \lambda f x \cdot f(f(x))\\ \dot{2} &= \lambda f x \cdot f(f(f(x)))\\ &\vdots\end{aligned}$$

Neither of these encodings is the most natural one. This is because both Church and Kleene worked under the λ-I-calculus. Most modern authors (for example, [2]) use the following:

$$\begin{aligned}\dot{0} &= \lambda f x \cdot x\\ \dot{1} &= \lambda f x \cdot f(x)\\ \dot{2} &= \lambda f x \cdot f(f(x))\\ &\vdots\end{aligned}$$

This final encoding is the one that I will use. In informal terms, the number n is identified with the formula representing a function which takes another function f as input, and outputs the n-fold composition of f with itself: $\dot{n} = f^{(n)}$. Note that the formula $\lambda f x \cdot x$, which represents $\dot{0}$, is a properly formed formula but not a well-formed formula. Therefore, this encoding requires that we are working under the λ-K-calculus, and not the λ-I-calculus.

4.4 Conversion and normal form

The power of the λ-calculus lies in what we can do with formulas:

> From [5]:
>
> I. To replace any part $\lambda\mathbf{x}[\mathbf{M}]$ of a formula by $\lambda\mathbf{y}[S^{\mathbf{x}}_{\mathbf{y}}\mathbf{M}|]$, where $\mathbf{y}$ is a variable which does not occur in $\mathbf{M}$.
>
> II. To replace any part $\{\lambda\mathbf{x}[\mathbf{M}]\}(\mathbf{N})$ of a formula by $S^{\mathbf{x}}_{\mathbf{N}}\mathbf{M}|]$, provided that the bound variables in $\mathbf{M}$ are distinct both from $\mathbf{x}$ and from the free variables in $\mathbf{N}$.
>
> III. To replace any part $S^{\mathbf{x}}_{\mathbf{N}}\mathbf{M}|]$ (not immediately following λ) of a formula by $\{\lambda\mathbf{x}[\mathbf{M}]\}(\mathbf{N})$, provided that the bound variables in $\mathbf{M}$ are distinct both from $\mathbf{x}$ and from the free variables in $\mathbf{N}$.

In these definitions, $S^{\mathbf{x}}_{\mathbf{N}}\mathbf{M}|$ refers to the string obtained by replacing every occurrence of $\mathbf{x}$ in $\mathbf{M}$ by $\mathbf{N}$.

Operation I can be thought of as renaming variables. In other words, it formalizes the tie between functions such as $x \mapsto x$ and $y \mapsto y$. Operations II and III correspond to going forward and backward, respectively, in function evaluation.

> From [5]:
>
> Any finite sequence of these operations is called a *conversion*, and if $\mathbf{B}$ is obtainable from $\mathbf{A}$ by a conversion we say that $\mathbf{A}$ is *convertible* into $\mathbf{B}$, or, "$\mathbf{A}$ conv $\mathbf{B}$."

Convertability is an equivalence relation on formulas in that it is reflexive, symmetric, and transitive. Also note that conversion preserves the property of being properly- or well-formed.

> From [5]:
>
> A conversion which contains exactly one application of Operation II, and no application of Operation III, is called a *reduction*.
>
> A formula is said to be in *normal form* if it is well-formed and contains no part of the form $\{\lambda\mathbf{x}[\mathbf{M}]\}(\mathbf{N})$.

Since we are working with properly formed formulas, the above definition should be altered to read "properly formed" in place of "well-formed." The normal form can be thought of as the final result of a computation, after all functions have been evaluated. A properly formed formula is said to have a normal form if it is convertible (actually, reducible) to a formula in normal form. Although not all properly formed formulas have a normal form, it is obvious that once a pff is in normal form, no further reduction of it is possible. Less obvious is that the normal form of a pff is unique up to renaming by Operation I, and that if a pff has a normal form, then any strategy of reduction will eventually find it [6].

Example 4. Consider the pff $\dot{\imath}(\dot{\imath})$. In the conversion steps below, underbraces mark the subformulas that appear in the definitions of the operations.

Step 1: $\dot{1}(\dot{1}) = \{\lambda \underbrace{f}_{\mathbf{x}}[\underbrace{\lambda x \cdot f(x)}_{\mathbf{M}}]\}(\underbrace{\lambda f x \cdot f(x)}_{\mathbf{N}})$ conv — by Operation II

Step 2: $\lambda x \cdot \{\lambda f[\underbrace{\lambda x \cdot f(x)}_{\lambda \mathbf{x}[\mathbf{M}]}]\}(x)$ conv — by Operation I

Step 3: $\lambda x \cdot \{\lambda \underbrace{f}_{\mathbf{x}}[\underbrace{\lambda y \cdot f(y)}_{\mathbf{M}}]\}(\underbrace{x}_{\mathbf{N}})$ conv — by Operation II

Step 4: $\lambda xy \cdot x(y)$ — Normal form

Note that in the above example, at step 2: $\lambda x \cdot \{\lambda f[\lambda x \cdot f(x)]\}(x)$, there is no way to proceed with reduction by Operation II because x is a free variable of the putative $\mathbf{N}$ and a bound variable of the putative $\mathbf{M}$. It is necessary to rename x to avoid this. There is no procedure for renaming free variables, so we rename the bound copy of x by Operation I. In general, if we have a formula of the form $\{\lambda \mathbf{x}[\mathbf{M}]\}(\mathbf{N})$ that we wish to reduce, and $\mathbf{x}$ or a free symbol of $\mathbf{N}$ is a bound symbol of $\mathbf{M}$, we can always apply Operation I to get $\{\lambda \mathbf{y}[S^{\mathbf{x}}_{\mathbf{y}}\mathbf{M}|]\}(\mathbf{N})$, choosing a $\mathbf{y}$ which does not already appear in $\mathbf{M}$ and which is not a free variable of $\mathbf{N}$. Operation II is then possible. Using this technique, reduction is always possible for a pff not already in normal form. However, it is not guaranteed that carrying out repeated reductions is a process that will necessarily terminate, as the reader can verify by attempting to reduce the following formula: $\{\lambda x \cdot x(x)\}(\lambda x \cdot x(x))$. In [5], Church famously showed that the problem of whether or not an arbitrary term has a normal form is undecideable.

4.5 λ-definable functions

Now consider arbitrary functions from n-tuples of natural numbers to the natural numbers. For any such function f, if it has a corresponding pff $\mathbf{F}$ in the λ-calculus, then it is called λ-*definable* and we say that $\mathbf{F}$ λ-defines f. By a corresponding formula, we mean that for any $x_1, ..., x_n \in \mathbb{N}$, we have $\mathbf{F}(\dot{\mathbf{x}}_\mathbf{1}, ..., \dot{\mathbf{x}}_\mathbf{n})$ conv $\dot{\mathbf{x}}$ where $\dot{\mathbf{x}}_\mathbf{1}, ..., \dot{\mathbf{x}}_\mathbf{n}$ are the Church numerals corresponding to the natural numbers $x_1, ..., x_n$ and $\dot{\mathbf{x}}$ is the Church numeral corresponding to the value of $f(x_1, ..., x_n)$.

The intuitive relationship between λ-definability and computability follows immediately from the properties of the λ-calculus. Recall that if a pff has a normal form, then any reduction strategy must eventually find it [6]. By definition, if a pff $\mathbf{F}$ λ-defines a function, then for any inputs $\dot{\mathbf{x}}_\mathbf{1}, ..., \dot{\mathbf{x}}_\mathbf{n}$, $\mathbf{F}(\dot{\mathbf{x}}_\mathbf{1}, ..., \dot{\mathbf{x}}_\mathbf{n})$ has a normal form, namely, the "value" $\dot{\mathbf{x}}$ which, as a Church numeral, is in normal form. Therefore, by selecting any strategy and consistently applying it, we have an algorithm for computing the values of a λ-definable function.

In light of the above, the λ-calculus is a model for computation in the same way that the Turing machine is. Kleene [15] remarked that the λ-calculus has "the remarkable feature that it is all contained in a very simply and almost inevitable formulation, arising in a natural connection with no prethought of the result. And a given λ-formula engenders the computation procedure for the function it defines."

5 Equivalence

Though it is apparent that the λ-calculus is well-motivated by the notion of computability, it seems somewhat mysterious that the λ-calculus and recursion are linked in any way. After all, it seems that in the λ-calculus, there is no way to define a function that calls itself, which constitutes an intuitive understanding of recursion. It is a remarkable fact that this actually *is* possible. In the present section, we explore the link between the notions of recursion from section 3 and the λ-calculus of section 4. The first proof of the equivalence of the general recursive functions and the λ-definable functions was sketched in [5] and fully developed in [13]. I will draw extensively from these papers. However, both papers worked under the λ-I-calculus. Both also mention that the corresponding result for the λ-K-calculus was obtained by Rosser, but this work does not seem to have been published. As mentioned previously, we will be working under the λ-K-calculus for simplicity of proof and exposition.

5.1 All μ-recursive functions are λ-definable

In order to show that all μ-recursive functions are λ-definable, we must prove that every μ-recursive function has a corresponding pff with the property described in section 4.5. We directly construct such pffs for the initial functions.

The corresponding pff for the successor function is $S = \lambda\rho fx \cdot f(\rho(f,x))$. To illustrate that this appropriately represents the successor function, consider the action of this formula on the Church numeral for 3:

$S(\dot{3}) = \{\lambda\rho fx \cdot f(\rho(f,x))\}(\lambda fx \cdot f(f(f(x)))) \text{ conv}$ by Operation II
$\lambda fx \cdot f(\{\lambda fx \cdot f(f(f(x)))\}(f,x)) \text{ conv}$ by Operation II
$\lambda fx \cdot f(\{\lambda x \cdot f(f(f(x)))\}(x) \text{ conv}$ by Operation II
$\lambda fx \cdot f(f(f(f(x)))) = \dot{4}$

The λ-definitions for the constant functions and the projection functions are equally simply under the λ-K-calculus. For the constant function $\varphi(x_1,...,x_n) = q$, the corresponding pff is $\lambda t_1...t_n \cdot \dot{\mathbf{q}}$. For the projection function $\varphi(x_1,...,x_n) = x_i$, the corresponding pff is $\lambda t_1...t_n \cdot t_i$. These definitions are immediate and natural. However, note that these formulas are not well-formed, only properly formed. Therefore, they are not admissible under the λ-I-calculus. Due to this limitation, Kleene was forced, in his original proof, to resort to more elaborate definitions. For example, his projection function is λ-defined by $\lambda t_1...t_n \cdot t_1(I,...,t_n(I,t_i)...)$ where I is the formula $\lambda f \cdot f$. In order to define the constant functions, one must compose $\lambda t \cdot t(I,\dot{0})$, which always returns $\dot{0}$, with the successor function and the projection functions.

By definition, a μ-recursive function is constructed from initial functions by applications of the schemata of substitution, primitive recursion, and minimization, which we may consider as operators that take in functions and produce a new one. Thus, we must show that each of these schemata is λ-definable in the sense that if the input functions are λ-definable, then the output function is also λ-definable. In the case of substitution, this is easily shown by construction.

To define a function by substitution we must already have on hand functions $\psi, \chi_1, ..., \chi_m$ which are known to be p.r. Similarly, in the λ-calculus, suppose you are

given properly formed formulas $\mathbf{X_1}, ..., \mathbf{X_m}$ which λ-define the functions $\chi_1, ..., \chi_m$, and $\mathbf{\Psi}$ which λ-defines ψ. By definition, for arbitrary $\dot{\mathbf{x}}_\mathbf{1}, ..., \dot{\mathbf{x}}_\mathbf{n}$, the pffs $\mathbf{X_i}(\dot{\mathbf{x}}_\mathbf{1}, ..., \dot{\mathbf{x}}_\mathbf{n})$ are convertible to Church numerals $\dot{\mathbf{y}}_\mathbf{i}$, and the pff $\mathbf{\Psi}(\dot{\mathbf{y}}_\mathbf{1}, ..., \dot{\mathbf{y}}_\mathbf{m})$ is convertible to a Church numeral $\dot{\mathbf{y}}$. We wish to construct a properly formed formula, which when applied to $\dot{\mathbf{x}}_\mathbf{1}, ..., \dot{\mathbf{x}}_\mathbf{n}$, is convertible to $\dot{\mathbf{y}}$.

The natural choice is $\lambda t_1 ... t_n \cdot \mathbf{\Psi}(\mathbf{X_1}(t_1, ..., t_n), ..., \mathbf{X_m}(t_1, ..., t_n))$.

$\{\lambda t_1 ... t_n \cdot \mathbf{\Psi}(\mathbf{X_1}(t_1, ..., t_n), ..., \mathbf{X_m}(t_1, ..., t_n))\}(\dot{\mathbf{x}}_\mathbf{1}, ..., \dot{\mathbf{x}}_\mathbf{n})$ conv

...by reducing n times...

$\mathbf{\Psi}(\mathbf{X_1}(\dot{\mathbf{x}}_\mathbf{1}, ..., \dot{\mathbf{x}}_\mathbf{n}), ..., \mathbf{X_m}(\dot{\mathbf{x}}_\mathbf{1}, ..., \dot{\mathbf{x}}_\mathbf{n}))$ conv

$\mathbf{\Psi}(\dot{\mathbf{y}}_\mathbf{1}, ..., \dot{\mathbf{y}}_\mathbf{1})$ conv

$\dot{\mathbf{y}}$ as required.

Finally, we require the λ-calculus equivalents of the primitive recursion operator and the minimization operator. These are not as easily obtained as the previous schemata. Kleene's proof is unwieldy, and can barely be considered constructive [13]. We give a more elegent exposition, similar to the one given in [1], by considering fixed points. We will further simplify the exposition by only considering functions of one variable.

In the context of the λ-calculus, a fixed point for a pff $\mathbf{F}$ is defined to be a pff $\mathbf{M}$ such that $\mathbf{F}(\mathbf{M})$ conv $\mathbf{M}$. For example, if $\mathbf{F}$ is $\lambda f \cdot f$, then every pff is a fixed point of $\mathbf{F}$. Remarkably, every pff has an easy-to-find fixed point. A *fixed-point combinator* is defined to be a pff $\mathbf{Y}$ such that for any pff $\mathbf{F}$, $\mathbf{F}(\mathbf{Y}(\mathbf{F}))$ conv $\mathbf{Y}(\mathbf{F})$. In other words, by applying $\mathbf{Y}$ to any $\mathbf{F}$, we get a fixed point for $\mathbf{F}$. Does such a $\mathbf{Y}$ exist? In fact, several have been found. According to [7], Turing was the first to discover one. His fixed-point combinator, which he called Θ, is given by $\{\lambda f x \cdot x(f(f(x)))\}(\lambda f x \cdot x(f(f(x))))$. An even simpler one by Haskell Curry, known as the Y combinator, is given by $Y = \lambda f \cdot \{\lambda x \cdot f(x(x))\}(\lambda x \cdot f(x(x)))$.

$Y(\mathbf{F}) = \{\lambda f \cdot \{\lambda x \cdot f(x(x))\}(\lambda x \cdot f(x(x)))\}(\mathbf{F})$ conv

$\{\lambda x \cdot \mathbf{F}(x(x))\}(\lambda x \cdot \mathbf{F}(x(x)))$ conv

$\mathbf{F}(\{\lambda x \cdot \mathbf{F}(x(x))\}(\lambda x \cdot \mathbf{F}(x(x))))$ conv

$\mathbf{F}(\{\lambda f \cdot \{\lambda x \cdot f(x(x))\}(\lambda x \cdot f(x(x)))\}(\mathbf{F})) = \mathbf{F}(Y(\mathbf{F}))$ as required.

Primitive recursion is a schema for defining a function. However, it does not quite specify an algorithm for *evaluating* that function in a way that a computer could follow. We can define such an algorithm as follows:

Require: χ, an $(n+1)$-place function
Require: ψ, an $(n-1)$-place function
  function $R(x_1, ..., x_n)$
    if $x_1 = 0$ **then return** $\psi(x_2, ..., x_n)$
    else return $\chi(x_1 - 1, R(x_1 - 1, x_2, ..., x_n), x_2, ..., x_n)$
    end if
  end function

The challenge in expressing the above algorithm in the λ-calculus is the restriction that a function cannot call itself directly. This is because functions in the λ-calculus

are "anonymous." In the above algorithm, we have given the name R to the function. However, in the λ-calculus, this is not possible. Although we have assigned names to certain pffs in previous examples, these are only abbreviations for our benefit. They have no status in the theory. In order to get around this, we use a higher-order encapsulating function.

Require: χ as above
Require: ψ as above
  function $R'(f)$
    return function $(x_1, ..., x_n)$
      if $x_1 = 0$ **then return** $\psi(x_2, ..., x_n)$
      else return $\chi(x_1 - 1, f(x_1 - 1, x_2, ..., x_n), x_2, ..., x_n)$
      end if
    end function
  end function

The function R' takes in a function f, and returns an anonymous function which depends on f. That anonymous function takes numbers $x_1, ..., x_n$. If $x_1 = 0$, the anonymous function returns $\psi(x_2, ..., x_n)$. Otherwise, it returns the result of $\chi(x_1 - 1, f(x_1 - 1, x_2, ..., x_n), x_2, ..., x_n)$. R' does not call itself, so some analogue of it should be λ-definable. Most importantly, R is a fixed point of R'. That is to say, $R'(R)$ returns a function that behaves exactly the same as R does.

Now, we translate the above into the λ-calculus. Define $T = \lambda xy \cdot x$ and $F = \lambda xy \cdot y$. If A conv T, then $A(B, C)$ conv B. If A conv F, then $A(B, C)$ conv C. In other words, $A(B, C)$ means "if A, then B, else, C," and T and F represent "true" and "false," respectively. Define $Z = \lambda x \cdot x(\lambda xyz \cdot z, T)$. Then $Z(\dot{0})$ conv T and for $\dot{\mathbf{x}}$ a Church numeral not $\dot{0}$, $Z(\dot{\mathbf{x}})$ conv F. In other words, $Z(\dot{\mathbf{x}})$ asks if $\dot{\mathbf{x}}$ is $\dot{0}$.

Now it only remains to define the predecessor function. An intuitive definition can be obtained by considering an ordered pair data structure [17]. The idea is, for any given n, to construct the Church numeral $\mathbf{n} \dot{-} \mathbf{1}$ by applying a modified successor formula n times. This modified successor formula simultaneously increments one element of a pair, and assigns the pre-incremented value to the other element. In other words, we use pairs to provide a "memory" location capable of storing one additional numeral. Then, after n applications, one element of the pair will have value $\dot{\mathbf{n}}$, while the other will have $\mathbf{n} \dot{-} \mathbf{1}$ since it lags behind.

For pffs $\mathbf{A}$, $\mathbf{B}$, let $\langle \mathbf{A}, \mathbf{B} \rangle = \lambda f \cdot f(\mathbf{A}, \mathbf{B})$. The first and second elements can be accessed by $\langle \mathbf{A}, \mathbf{B} \rangle(T)$ conv $\mathbf{A}$ and $\langle \mathbf{A}, \mathbf{B} \rangle(F)$ conv $\mathbf{B}$. Define $\Sigma = \lambda x \cdot \langle x(F), S(x(F)) \rangle$. This is our modified successor formula. If n is a natural number, $\dot{\mathbf{n}}(\Sigma)$ represents Σ applied n times. Then $\{\dot{\mathbf{n}}(\Sigma)\}(\langle \dot{0}, \dot{0} \rangle)$ returns $\langle \mathbf{n} \dot{-} \mathbf{1}, \dot{\mathbf{n}} \rangle$, so the answer that we want is $\dot{\mathbf{n}}(\Sigma, \langle \dot{0}, \dot{0} \rangle, F)$. Therefore, the predecessor function is λ-defined by the pff $P = \lambda x \cdot x(\Sigma, \langle \dot{0}, \dot{0} \rangle, F)$.

Now we have all of the components necessary to assemble a definition by primitive recursion. If χ and ψ are as in the primitive recursion schema (Vb), and are λ-defined by $\mathbf{X}$ and $\mathbf{\Psi}$, then the resulting φ is λ-defined by:

$$Y(\lambda f t_1 \cdots t_n \cdot Z(t_1, \mathbf{\Psi}(t_2, ..., t_n), \mathbf{X}(P(t_1), f(P(t_1), t_2, ..., t_n), t_2, ..., t_n)))$$

We motivate the minimization operator by presenting an algorithm similar to the one presented for primitive recursion:

Require: χ, an $(n+1)$-place predicate such that $(\forall x_1 \cdots x_n)(\exists y)[\chi(x_1, ..., x_n, y) = 0]$
  function $M(x_1, ..., x_n, y)$
    if $\chi(x_1, ..., x_n, y) = 0$ **then return** y
    else return $M(x_1, ..., x_n, y + 1)$
    end if
  end function

Again, to get around the problem of a function calling itself when it has no name in the λ-calculus, we define an encapsulating function:

Require: χ as above
  function $M'(f)$
    return function $(x_1, ..., x_n, y)$
      if $\chi(x_1, ..., x_n, y) = 0$ **then return** y
      else return $f(x_1, ..., x_n, y + 1)$
      end if
    end function
  end function

Our desired function is a fixed point of M', evaluated at $y = 0$. Therefore, if χ is as in the minimization schema (VIb), and it is λ-defined by $\mathbf{X}$, then the resulting φ is λ-defined by:

$$\{Y(\lambda f y t_1 ... t_n \cdot Z(\mathbf{X}(t_1, ..., t_n, y), y, f(t_1, ..., t_n, S(y))))\}(\dot{0})$$

5.2 All λ-definable functions are μ-recursive

Conversely, the proof that all λ-definable functions are μ-recursive is very similar to the proof of the Kleene Normal Form Theorem from section 3.3. In fact, we might say that it is analogous:

- The formal system of the λ-calculus will play the part of the formal system for recursive functions in the Herbrand-Gödel definition.
- The rules R1 and R2 are replaced by the Operations I, II, III.
- Almost every step of the proof will be seen to be analogous.
- The final goal is to produce the same decomposition for an arbitrary λ-computable function as was obtained for an arbitrary general recursive function in the Kleene Normal Form Theorem.

In order to be able to reuse the p.r. functions defined in the proof of the Kleene Normal Form Theorem, we use the Gödel numbering given in the table below. The GN in section 3.3 was similarily chosen so as to be able to reuse the functions from the original paper of Gödel [11].

Symbol	λ	$\{, (, [$	$\},),]$	the i^{th} variable
Gödel number	1	11	13	p_{i+6}

As formulas in the λ-calculus are analogous to expressions in the Herbrand-Gödel definition of recursion, we assign GNs to formulas, sequences of formulas, etc. (in general, "items") in the same way as before.

The rest of the proof is given as a sketch, with reference to the analogous steps in the proof of the Kleene Normal Form Theorem:

Herbrand-Gödel definition	λ-calculus
• Suppose f is recursively defined by the system of equations $\mathbf{E}$, and σ represents f in $\mathbf{E}$.	• Suppose f is λ-defined by the pff $\mathbf{F}$.
• For any set of natural numbers $x_1, ..., x_n$, the equation $\sigma(\dot{\mathbf{x}}_\mathbf{1}, ..., \dot{\mathbf{x}}_\mathbf{n})) = \dot{\mathbf{x}}$ is obtainable by applications of the rules R_i iff $f(x_1, ..., x_n) = x$.	• For any set of natural numbers $x_1, ..., x_n$, $\mathbf{F}(\dot{\mathbf{x}}_\mathbf{1}, ..., \dot{\mathbf{x}}_\mathbf{n})$ conv $\dot{\mathbf{x}}$ if and only if $f(x_1, ..., x_n) = x$.
• Construct a p.r. function *Rule* corresponding to the rules R_i.	• Construct a p.r. function *Op* corresponding to the Operations I, II, III.
• Construct a p.r. function C that enumerates all numbers (GNs of expressions) obtainable by applying *Rule* to GNs of components of a system of equations.	• Construct a p.r. function C^λ that enumerates all numbers (GNs of formulas) obtainable by applying *Op* to GNs of a pff.
• From C, construct a p.r. predicate $E(x_1, ..., x_n, y)$ and a p.r. function $V(y)$.	• From C^λ, construct a p.r. predicate $E^\lambda(x_1, ..., x_n, y)$ and a p.r. function $V^\lambda(y)$.
• As y varies, $E(x_1, ..., x_n, y)$ runs through the expressions obtainable from $\mathbf{E}$ and checks to see if they are of the form $\sigma(\dot{\mathbf{x}}_\mathbf{1}, ..., \dot{\mathbf{x}}_\mathbf{n})) = \dot{\mathbf{x}}$ for some numeral $\dot{\mathbf{x}}$.	• As y varies, $E^\lambda(x_1, ..., x_n, y)$ runs through the formulas convertible to $\mathbf{F}(\dot{\mathbf{x}}_\mathbf{1}, ..., \dot{\mathbf{x}}_\mathbf{n})$ and checks to see if they are Church numerals.
• For y such that $E(x_1, ..., x_n, y)$ returns true, $V(y)$ returns x, the natural number corresponding to the numeral $\dot{\mathbf{x}}$ (not the GN).	• For y such that $E^\lambda(x_1, ..., x_n, y)$ returns true, then $V^\lambda(y)$ returns the natural number corresponding to the Church numeral found.
• Conclude: $f(x_1, ..., x_n) = V(\mu y E(x_1, ..., x_n, y))$	• Conclude: $f(x_1, ..., x_n) = V^\lambda(\mu y E^\lambda(x_1, ..., x_n, y))$

The details are left to the reader.

6 The Church-Turing thesis

In [5], Church first proposed a version of what is now known as Church's thesis, or the Church-Turing thesis, now one of the most famous statements in Computer Science. The statement says that the λ-definable functions, the Turing computable functions, and the general recursive functions, which are all the same, are exactly those functions that are "effectively calculable" or "effectively computable." Church defined an effectively calculable function as one for which an algorithm exists to calculate its values. Without a mathematical definition of "algorithm," this is an intuitive notion, and therefore the thesis is "not susceptible of proof" [18]. However, as noted in section 4.5, there certainly

seems to be some intuitive justification for the thesis in the case of the λ-definable functions.

Both Church and Turing proposed the identification of effective computability with their respective systems not as a "thesis," but as a definition. This is certainly the most sane way to regard it. However, since then, the Church-Turing thesis has sometimes been invoked as more of a lemma [8]. In certain proofs, a set may be shown to be recursively enumerable by stating an algorithm that decides whether or not something is a member of it, then invoking the Church-Turing thesis. The utility of the thesis in simplifying proofs in this way has contributed to its perceived importance. However, the most important implication of the Church-Turing thesis is that there exist problems that are *not* solvable by algorithm. We have already discussed one such problem, Hilbert's *Entscheidungsproblem*. Thanks to the work of Church and Turing, we know that no algorithm can decide whether or not an arbitrary mathematical statement is a theorem. Thus, there is an absolute limit to what computers can accomplish, or what recursion can express.

On the other hand, the Church-Turing thesis also implies that a system even as simple as the λ-calculus can theoretically express any possible algorithm. Furthermore, an enormous number of interesting and independently motivated definitions have been shown to be equivalent to the ones discussed in the present paper [14]. That such connections exist is often remarkable. Even *a posteriori*, it seems strange that recursive functions can be expressed in the λ-calculus, which seems to lack any notion of recursion. Nevertheless, we saw that this is true, and almost trivial. From this, it is easy to imagine that there may be something more to the Church-Turing thesis than just definition.

References

[1] H. Barendregt. The type free lambda calculus. In J. Barwise, editor, *Handbook of Mathematical Logic*. North Holland, 1981.

[2] H. Barendregt and E. Barendsen. Introduction to lambda calculus. `http://www.cse.chalmers.se/research/group/logic/TypesSS05/Extra/geuvers.pdf`.

[3] H.P. Barendregt. *The lambda calculus: its syntax and semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North Holland, 1984.

[4] A. Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932.

[5] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

[6] A. Church and J.B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, 1936.

[7] H.B. Curry and R. Feys. *Combinatory Logic*, volume 2. North-Holland, 1972.

[8] D. Dalen. Algorithms and decision problems: A crash course in recursion theory. In D. M. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume 1, pages 245–312. Kluwer Academic Publishers, 2nd edition, 2001.

[9] K. Gödel. Über formal unentscheidbare sätze der Principia mathematica und verwandter Systeme I. *Monatshefte für Mathematik*, 38(1):173–198, 1931.

[10] K. Gödel. On undecidable propositions of formal mathematical systems. In M. Davis, editor, *The Undecidable, Basic Papers on Undecidable Propositions, Unsolvable Problems And Computable Functions*. Raven Press, 1965. A transcription of a 1934 lecture by Gödel at Princeton.

[11] K. Gödel. On formally undecidable propositions of Principia Mathematica and related systems I. In S. Feferman, editor, *Kurt Gödel Collected Works*, volume 1. Oxford University Press, 1986. translation of [9].

[12] S.C. Kleene. General recursive functions of natural numbers. *Mathematische Annalen*, 112(1):727–742, 1936.

[13] S.C. Kleene. λ-definability and recursiveness. *Duke Mathematical Journal*, 2(2):340–353, 1936.

[14] S.C. Kleene. *Introduction to Metamathematics*. Van Nostrand, 1952.

[15] S.C. Kleene. Origins of recursive function theory. *Annals of the History of Computing*, 3(1):52–67, 1981.

[16] S.C. Kleene and J.B. Rosser. The inconsistency of certain formal logics. *Annals of Mathematics*, 36(3):630–636, 1935.

[17] R. Rojas. A tutorial introduction to the lambda calculus. `http://www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf`.

[18] J.B. Rosser. Highlights of the history of the lambda-calculus. In *Proceedings of the 1982 ACM symposium on LISP and functional programming*, pages 216–225. ACM, 1982.

[19] R.I. Soare. Computability and recursion. *Bulletin of Symbolic Logic*, pages 284–321, 1996.

[20] A.M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, Series 2, 42:230–265, 1937.