An Introduction to the Lambda Calculus

Mayer Goldberg

February 20, 2000

1 Notation and Conventions

It is surprising that despite the simplicity of its syntax, the λ-calculus hosts a large body of notation, abbreviations, naming conventions, etc. Our aim, as far as the notation throughout this work is concerned, is to remain consistent, clear and unambiguous as much as possible. By and large we adhere to the notation set down in Church's *Calculi of Lambda Conversion* [4], and Barendregt's *The Lambda Calculus, Its Syntax and Semantics* [1]. There are however instances where our notation diverges. Most notably, we avoid abbreviating the names of combinators. Thus, for example, we use $\mathbf{Succ}_{\mathsf{Church}}$ to denote the λ-term computing the successor function on Church numerals, instead of Church's S^+,[1] etc.

A variety of typefaces are used in denoting logical and mathematical entities. In general, we strive for consistency, unless prevented from doing so by existing, well-established notation.

The following table illustrates the various typefaces used throughout this work, and the context in which they appear:

Typeface Usage within this Document	
Math Italics	variables, λ-terms used locally
Roman Bold	λ-terms used throughout this work
San Serif	sets, relations, trees, syntactic objects
Fraktur Gothic: $\mathfrak{Fraktur}$	arithmetic functions
Greek: α, β, γ, ...	rules, well-known combinators
Blackboard bold: $\mathbb{N}$, $\mathbb{Q}$, $\mathbb{Z}$, ...	well-known sets of numbers
$\mathcal{CALLIGRAPHIC}$	systems of equations

Our introduction to the λ-calculus consists of five sections, which cover the following topics, in order: *Syntax, reduction, λ-definability, fixed-points,* and *bases*. By no stretch of the imagination can this selection of topics be considered a complete survey of the main areas of the λ-calculus. The most striking omissions include *models* and *types*. This introduction is, however, sufficient for its intended purposes, which are, as stated earlier, to make this work as self-contained as possible, and to establish a common language with the reader, in terms of which, the ideas contained in this work can be conveyed.

The sections on syntax and reduction cover the λ-calculus as a formalism for describing computation. Section 4 covers λ-definability, which concerns itself with representing logical and mathematical entities in the λ-calculus. Section 5 covers fixed points. The treatment is not complete, but is sufficient for our purposes. Section 6 concludes the introduction to the λ-calculus with a treatment of bases. Bases in the λ-calculus are analogous, in many ways, to bases in linear algebra.

[1] The latter, S^+, could easily be confused for the set of λ-terms *generated* by the set S, in accordance with Definition 2.16, which follows Barendregt's text [1, Definition 8.1.1, Page 165].

2 Syntax

2.1 Lambda Terms

The domain of discourse in the λ-calculus is a set Λ of terms, known as "λ-terms" or "λ-expressions." The construction of the set Λ follows.

2.1 Definition: *The sets* Symbols *and* Λ. Let Symbols be a countable set of symbols. The set Λ of all λ-terms is defined inductively as follows:

i. $\mathsf{Symbols} \subset \Lambda$

ii. For any $M, N \in \Lambda$, we have $(M\ N) \in \Lambda$. This syntactic form is called an *application.*

iii. For any $M \in \Lambda$, $\nu \in \mathsf{Symbols}$, we have $(\lambda\nu.M) \in \Lambda$. This syntactic form is known as λ-*abstraction,* (or just plain *"lambda"*). We refer to M in the context of the λ-abstraction as the *body* of the *lambda.*

The set Λ is also known by $\Lambda_{\mathbf{K}}$, distinguishing it from a slightly different set of λ-terms, the $\Lambda_{\mathbf{I}}$. The set $\Lambda_{\mathbf{I}}$ will be introduced in Definition 2.14 after we define the notions of *free* and *bound* variables.

2.2 Intuition: It is often useful to think of λ-abstraction and application as modelling function abstraction and the application of a function to an argument. In this work, however, this modelling is not taken to be more than a useful intuition.

2.3 Conventions:

i. We may remove any pair of parentheses or even the dot separator between the variable and the body in a λ-abstraction, whenever this cannot result in ambiguity. For example: The λ-term $(\lambda x.x)$ by itself could appear as $\lambda x.x$, but in an application, we keep the parentheses: $\lambda y.((\lambda x.x)y)$.

ii. For any $M_1, M_2, M_3 \in \Lambda$, the λ-term $((M_1\ M_2)\ M_3)$ can be abbreviated as $(M_1\ M_2\ M_3)$. This convention is known as the *left-associativity* of application.

iii. For any $\nu_1, \nu_2 \in \mathsf{Symbols}, M \in \Lambda$, the λ-term $\lambda\nu_1.\lambda\nu_2.M$ can be abbreviated as $\lambda\nu_1\nu_2.M$.

2.4 Examples:

- The expression $(\lambda x.x)$ can in fact be abbreviated as λxx [4, Chapter I, Section 4, Page 7], but we avoid this abbreviation in the remainder of the work.
- The expression $(\lambda x.(\lambda y.(\lambda z.((x\ z)\ (y\ z)))))$ can be abbreviated as $\lambda xyz.(x\ z\ (y\ z))$.[2]

2.5 Definition: *Repeated Left- and Right-Associated Applications.* (See [1, Page 25, Definition 2.1.9]) Let $M, N \in \Lambda$, and $k \in \mathbb{N}$. Then

$$(M^k\ N) = \underbrace{(M \cdots (M}_{k} N) \cdots) \tag{1}$$

$$(M\ N^{\sim k}) = (M \underbrace{N \cdots N}_{k}) \tag{2}$$

2.6 Definition: *The length of a λ-term.* The *length* of a λ-term M, denoted by $\|M\|$, is defined inductively as follows:

$\|\nu\|$	$=$	1	where $\nu \in \mathsf{Symbols}$
$\|(M\ N)\|$	$=$	$1 + \|M\| + \|N\|$	where $M, N \in \Lambda$
$\|\lambda\nu.M\|$	$=$	$1 + \|M\|$	where $\nu \in \mathsf{Symbols}$, $M \in \Lambda$

[2] Terms in the literature are usually abbreviated a step further, by removing the parentheses around the body of the λ-abstraction, and compressing all spaces. For example, the λ-term $\lambda xyz.(x\ z\ (y\ z))$ can also be written as $\lambda xyz.xz(yz)$. This last abbreviation, however, can be quite confusing when symbol names are longer than one character. We therefore avoid such abbreviations throughout this work, so as to be able to use long, expressive symbol names.

2.7 Examples:

$$\begin{aligned} \|\lambda x.x\| &= 2 \\ \|\lambda x.(x\ x)\| &= 4 \\ \|\lambda xyz.(x\ z\ (y\ z))\| &= 10 \end{aligned}$$

2.8 Notation: *Arb.* A λ-term whose value is irrelevant is denoted by $\oslash$ (pronounced "arb", short for "arbitrary").

Arb is used in order to make arbitrary choices of λ-terms explicit.

The length of an arbitrary λ-term, $\|\oslash\|$, is an arbitrary positive integer. $\|\oslash\|$ is useful when reasoning about complexity.

2.2 Variables

2.9 Definition: *Set of Variables.* For any λ-expression M, the set $\mathsf{Vars}(M)$ of *variables* of M is defined inductively as follows. Let $\nu \in \mathsf{Symbols}$ and $M, N \in \Lambda$:

$$\begin{aligned} \mathsf{Vars}(\nu) &= \{\nu\} \\ \mathsf{Vars}((M\ N)) &= \mathsf{Vars}(M) \cup \mathsf{Vars}(N) \\ \mathsf{Vars}((\lambda\nu.M)) &= \mathsf{Vars}(M) \end{aligned}$$

2.10 Definition: *Set of Free Variables.* For any λ-expression M, the set $\mathsf{FreeVars}(M)$ of *free variables* of M is defined inductively as follows. Let $\nu \in \mathsf{Symbols}$ and $M, N \in \Lambda$:

$$\begin{aligned} \mathsf{FreeVars}(M) &= \mathsf{FreeVars}'(M, \emptyset) \\ \mathsf{FreeVars}'(\nu, S) &= \begin{cases} \{\nu\} & \text{if } \nu \notin S \\ \emptyset & \text{if } \nu \in S \end{cases} \\ \mathsf{FreeVars}'((M\ N), S) &= \mathsf{FreeVars}'(M, S) \cup \mathsf{FreeVars}'(N, S) \\ \mathsf{FreeVars}'(\lambda\nu.M, S) &= \mathsf{FreeVars}'(M, S \cup \{\nu\}) \end{aligned}$$

2.11 Definition: *Set of Bound Variables.* For any λ-expression M, the set $\mathsf{BoundVars}(M)$ of *bound variables* of M is defined inductively as follows. Let $\nu \in \mathsf{Symbols}$ and $M, N \in \Lambda$:

$$\begin{aligned} \mathsf{BoundVars}(\nu) &= \emptyset \\ \mathsf{BoundVars}((M\ N)) &= \mathsf{BoundVars}(M) \cup \mathsf{BoundVars}(N) \\ \mathsf{BoundVars}(\lambda\nu.M) &= \{\nu\} \cup \mathsf{BoundVars}(M) \end{aligned}$$

2.12 Examples:

$$\begin{aligned} \mathsf{Vars}(\lambda xyz.(x\ z\ (y\ z))) &= \{x, y, z\} \\ \mathsf{Vars}(\lambda xy.x) &= \{x\} \\ \mathsf{FreeVars}(\lambda x.(x\ y\ z)) &= \{y, z\} \\ \mathsf{BoundVars}(\lambda xw.(x\ y\ z)) &= \{x, w\} \end{aligned}$$

Note that

- As the last example shows, a variable can be contained in the set of bound variables of some expression, without actually *occuring* in that expression.
- A variable can have both free and bound occurrences within the same expression. For example:

$$\begin{aligned} \mathsf{FreeVars}((x\ (\lambda x.x))) &= \mathsf{BoundVars}((x\ (\lambda x.x))) \\ &= \{x\} \end{aligned}$$

When it is important to consider the free variables in a λ-term, we will tag a term with a subscript consisting of the free variables in the term. For example: $M_{x,y} = (x\ (x\ y))$.

2.13 Terminology: *A Fresh Variable.* Given a λ-term $M \in \Lambda$, a variable ν is said to be a *fresh variable* [for, or with respect to, M] if $\nu \notin \mathsf{Vars}(M)$.

2.14 Definition: *The set $\Lambda_{\mathbf{I}}$.* The set $\Lambda_{\mathbf{I}}$ of all $\lambda\mathbf{I}$-terms, is defined inductively as follows: Let $\mathsf{Symbols}$ be a countable set of symbols.

i. $\mathsf{Symbols} \subset \Lambda_{\mathbf{I}}$

ii. For any $M, N \in \Lambda$, we have $(M\ N) \in \Lambda_{\mathbf{I}}$.

iii. For any $M \in \Lambda_{\mathbf{I}}$, $\nu \in \mathsf{Symbols} \cap \mathsf{FreeVars}(M)$, we have $(\lambda\nu.M) \in \Lambda_{\mathbf{I}}$.

The set $\Lambda_{\mathbf{K}}$ (or simply Λ) is the domain of discourse of the $\lambda\mathbf{K}$-calculus (or simply the λ-calculus). The set $\Lambda_{\mathbf{I}}$ is the domain of discourse of the $\lambda\mathbf{I}$-calculus. These two calculi are defined in Definition 3.7.

2.3 Sequences

The following material is adapted from Barendregt's text on the λ-calculus [1, Chapter 2, Item 2.1.3, Page 22]. These conventions help avoid much of the clutter that results from excessive use of meta-syntactic ellipses within λ-terms:

2.15 Abbreviation: *Sequences of Symbols, Sequences of λ-Terms.* When the size n of a finite sequence $\nu_1, \ldots, \nu_n$ of symbols, or $M_1, \ldots, M_n$ of λ-terms is known, or can be inferred from the context, or is simply irrelevant, the sequence can be abbreviated as $\vec{\nu}$ or $\vec{M}$ respectively.

2.16 Definition: *Set of λ-Terms Generated by a Given Set.* [1, Section 8.1, Definition 8.1.1] Let $\mathsf{S} \subseteq \Lambda$. The set of all λ-terms generated by S, is denoted by S^+ and is the smallest set W which satisfies

i. $\mathsf{S} \subseteq \mathsf{W}$

ii. For $M, N \in \mathsf{W}$, $(M\ N) \in \mathsf{W}$

Another way to view the set of λ-terms S^+ generated by a set S is that S^+ is the *closure* of S under application.

2.4 Combinators

2.17 Definition: *Combinator. Set of All Combinators.* A *combinator* is a λ-term $M \in \Lambda$ that contains no free variables, i.e., for which

$$\mathsf{FreeVars}(M) = \emptyset \tag{3}$$

The set of all combinators is denoted by Λ^0.

2.18 Definition: *Proper Combinator.* A *proper combinator* is an expression of the form $\lambda\vec{x}.M$, where $M \in \{\vec{x}\}^+$. The *arity* of a proper combinator is $|\{\vec{x}\}|$. The λ-term M in the context of the proper combinator, is referred to as the *body* of the proper combinator.

2.19 Examples:

i. The following λ-terms are proper combinators: $\lambda x.x$, $\lambda xy.x$, and $\lambda xy.(x\ y\ y)$.

ii. The following λ-terms are not proper combinators: $\lambda x.(x\ (\lambda x.x))$, and $\lambda x.(x\ y)$.

iii. The arity of $\lambda xy.y$ is 2.

The combinators listed below are used throughout this work, and appear in much of the literature on the λ-calculus:

$$\begin{aligned}
\omega &= \lambda x.(x\ x) && (4)\\
\mathbf{B} &= \lambda xyz.(x\ (y\ z)) && (5)\\
\mathbf{C} &= \lambda xyz.(x\ z\ y) && (6)\\
\mathbf{I} &= \lambda x.x && (7)\\
\mathbf{J} &= \lambda xyzt.(x\ y\ (x\ t\ z)) && (8)\\
\mathbf{K} &= \lambda xy.x && (9)\\
\mathbf{S} &= \lambda xyz.(x\ z\ (y\ z)) && (10)\\
\mathbf{U}_k^n &= \lambda x_0 \cdots x_n.x_k && (11)\\
\mathbf{W} &= \lambda xy.(x\ y\ y) && (12)\\
\mathbf{Y}_{\mathsf{Curry}} &= \lambda f.((\lambda x.(f\ (x\ x))) \\
&\qquad (\lambda x.(f\ (x\ x)))) && (13)\\
\mathbf{Y}_{\mathsf{Turing}} &= ((\lambda fx.(f\ (x\ x\ f))) \\
&\qquad (\lambda fx.(f\ (x\ x\ f)))) && (14)
\end{aligned}$$

2.5 Substitution

The substitution in a λ-term M, of a free variable ν by a λ-term N, is denoted in the literature by either $M[N/\nu]$ or $M[\nu := N]$. While it is important to recognise both notations, we use the latter in this work, because it is used in Barendregt's text on the λ-calculus [1, Item 2.1.15, Page 27].

2.20 Definition: *Substitution of Free Variables.* The substitution of *all* free occurrences of a variable ν in a λ-term M by a λ-term N is denoted by $M[\nu := N]$ and is defined inductively as follows:

$$\begin{aligned}
\nu'[\nu := N] &= \begin{cases} N & \text{if } \nu' = \nu \\ \nu' & \text{if } \nu' \neq \nu \end{cases} && (15)\\
(M'\ M'')[\nu := N] &= (M'[\nu := N]\ M''[\nu := N]) && (16)\\
(\lambda\nu.M')[\nu := N] &= \begin{cases} \lambda\nu'.M' & \text{where } \nu = \nu' \\ \lambda\nu'.M'[\nu := N] & \\ \quad \text{where } \mathsf{BoundVars}(M') \cap \mathsf{FreeVars}(N) = \emptyset & \end{cases} && (17)
\end{aligned}$$

The above definition does not specify the value of $M[\nu := N]$, where N contains free variables that are bound in M. Removing the restriction in (17) would violate the common mathematical intuition, according to which, an occurrence of a given variable is a *place holder* for whatever is substituted for that variable: We expect the property of being a place holder for a *given* variable to be invariant under substitution, for all variables *other than* the variable being substituted for. For example, suppose we relaxed the condition in (17), and allowed $\mathsf{BoundVars}(M') \cap \mathsf{FreeVars}(N) \neq \emptyset$: Consider the expression

$$\lambda x.(\lambda y.\lambda x.y)[y := x] \tag{18}$$

The variable being substituted for is y, so the property of being a place holder for the variable in the outer 'λx' should be invariant under substitution, and yet, the result of the substitution would be $\lambda x.\lambda x.x$, and the occurrence of x is no longer a place holder for the variable in the outer 'λx', but rather for the inner one instead.

The problem of specifying $(\lambda\nu.M')[\nu := N]$, where $\mathsf{BoundVars}(M') \cap \mathsf{FreeVars}(N) \neq \emptyset$, is dealt with completely by the following convention: That an occurrence of a variable in a λ-term is simply a place holder for what is substituted for that variable, is a relationship between a variable occurrence and the 'λ' that abstracts over this variable. Furthermore, this relationship is a property of the graph of the term rather than of the particular names we happen to choose in order to denote this relationship. Now that the names of variables are starting to get in the way of our mathematical intuition, we define the problem away by partitioning the set of λ-terms modulo the renaming of bound variables, i.e., the substitution of bound variable names for fresh names (see Terminology 2.13). The result of this partitioning is that terms such as $(\lambda x.x)$ and $(\lambda y.y)$ are considered to be the same.

Getting back to the situation that prompted this discussion, concerning the substitution of $(\lambda\nu.M')[\nu := N]$, where $\mathsf{BoundVars}(M') \cap \mathsf{FreeVars}(N) \neq \emptyset$, it should be clear why this situation no longer poses a problem: The term $(\lambda\nu.M')$ is equivalent, in the sense we just discussed, to a term $(\lambda\nu.M'')$ that differs from the previous term in that all the variables $\mathsf{BoundVars}(M') \cap \mathsf{FreeVars}(N)$ which are bound in M' have been renamed to fresh variables, which do not occur in M' or N. We now have $\mathsf{BoundVars}(M'') \cap \mathsf{FreeVars}(N) = \emptyset$, and the substitution $(\lambda\nu.M'')[\nu := N]$ can be computed according to Definition 2.20. In light of the notion of equivalence modulo renaming of bound variables, the result of the substitution $(\lambda\nu.M'')[\nu := N]$ is taken as the result of $(\lambda\nu.M')[\nu := N]$ as well.

This notion of equivalence is known, in various forms, as α-rules:

2.21 Definition: *α-Conversion, α-Equivalence.*

- *α-Conversion.*

$$\lambda x.M \quad = \quad \lambda\nu.M[x := \nu] \tag{19}$$

  where $M \in \Lambda$, and ν is *fresh.*

- *α-Equivalence.* The set of λ-terms is partitioned modulo α-conversion, and so λ-terms that α-convert to each other are taken to represent the same entity.

In light of this discussion, α-conversion and α-equivalence will be implicit throughout the remainder of this work.

3 Reduction

3.1 Definition: *R-Redex, $\longrightarrow_{\mathsf{R}}$, $\twoheadrightarrow_{\mathsf{R}}$, $=_{\mathsf{R}}$, R-normal form (R-nf), The Diamond Property, Church-Rosser (CR).* [1, Definitions 3.1.3, 3.1.8, Page 51] Let R be a binary relation on Λ.

- An R-redex is a term $M \in \Lambda$, such that $\langle M, N\rangle \in \mathsf{R}$, for some term $N \in \Lambda$.
- $\longrightarrow_{\mathsf{R}}$ is the *one-step reduction:* For any $M, N \in \Lambda$, we have $M \longrightarrow_{\mathsf{R}} N$ if and only if $\langle M, N\rangle \in \mathsf{R}$.
- $\twoheadrightarrow_{\mathsf{R}}$ is the reflexive and transitive closure of $\longrightarrow_{\mathsf{R}}$.
- $=_{\mathsf{R}}$ is the equivalence relation generated by $\twoheadrightarrow_{\mathsf{R}}$.
- A λ-term $M \in \Lambda$ is R-nf if it contains no R-redex as a sub-expression. A λ-term N is R-nf of a λ-term M, if $M =_{\mathsf{R}} N$, and N is R-nf.
- Let R be a binary relation on Λ. R satisfies the *diamond property*, if for all $M, M_1, M_2 \in \Lambda$, such that $\langle M, M_1\rangle, \langle M, M_2\rangle \in \mathsf{R}$, we have $\langle M_1, M_3\rangle, \langle M_2, M_3\rangle \in \mathsf{R}$, for some $M_3 \in \Lambda$.
- *Church-Rosser (CR).* A relation R is CR, if $\twoheadrightarrow_{\mathsf{R}}$ satisfies the diamond property.

3.2 Definition: *The Relations $\twoheadrightarrow^{+}_{\mathsf{R}}$, $=^{+}_{\mathsf{R}}$.* Let R be a binary relation on Λ.

- $\longrightarrow\!\!\!\!\!\rightarrow_{\mathsf{R}}^{+}$ is the transitive closure of $\longrightarrow_{\mathsf{R}}$.
- For any λ-term $M, N \in \Lambda$, if $M =_{\mathsf{R}}^{+} N$ then there exists a λ-term $P \in \Lambda$, such that either

$$M \longrightarrow\!\!\!\!\!\rightarrow_{\mathsf{R}}^{+} P, \text{ and } N \longrightarrow\!\!\!\!\!\rightarrow_{\mathsf{R}} P$$

or

$$M \longrightarrow\!\!\!\!\!\rightarrow_{\mathsf{R}} P, \text{ and } N \longrightarrow\!\!\!\!\!\rightarrow_{\mathsf{R}}^{+} P$$

3.3 Observation: For any $M, N \in \Lambda$, and a binary relation R, if $M \longrightarrow\!\!\!\!\!\rightarrow_{\mathsf{R}} N$ then $M =_{\mathsf{R}} N$, and if $M \longrightarrow\!\!\!\!\!\rightarrow_{\mathsf{R}}^{+} N$ then $M =_{\mathsf{R}}^{+} N$. The converse does not always hold. In this sense, $\longrightarrow\!\!\!\!\!\rightarrow_{\mathsf{R}}$ and $\longrightarrow\!\!\!\!\!\rightarrow_{\mathsf{R}}^{+}$ are stronger properties than $=_{\mathsf{R}}$ and $=_{\mathsf{R}}^{+}$, respectively.

3.4 Definition: *The Binary Relations $\boldsymbol{\beta}$, $\boldsymbol{\eta}$, and $\boldsymbol{\beta\eta}$.* The binary relations $\boldsymbol{\beta}$, $\boldsymbol{\beta}$, and $\boldsymbol{\beta\eta}$ on Λ are defined as follows:

$$\boldsymbol{\beta} = \{\langle((\lambda x.M)\ N), M[x := N]\rangle : M, N \in \Lambda, x \in \mathsf{Symbols}\} \tag{20}$$

$$\boldsymbol{\eta} = \{\langle(\lambda x.(M\ x)), M\rangle : M \in \Lambda, x \in \mathsf{Symbols} - \mathsf{FreeVars}(M)\} \tag{21}$$

$$\boldsymbol{\beta\eta} = \boldsymbol{\beta} \cup \boldsymbol{\eta} \tag{22}$$

3.5 Abbreviations: *The Relations $\longrightarrow$, $\longrightarrow\!\!\!\!\!\rightarrow$, $=$, $\longrightarrow\!\!\!\!\!\rightarrow^{+}$, $=^{+}$.* Throughout the remainder of this work, the following abbreviations are used:

- $\longrightarrow$ abbreviates $\longrightarrow_{\beta\eta}$.
- $\longrightarrow\!\!\!\!\!\rightarrow$ abbreviates $\longrightarrow\!\!\!\!\!\rightarrow_{\beta\eta}$.
- $=$ abbreviates $=_{\beta\eta}$.
- $\longrightarrow\!\!\!\!\!\rightarrow^{+}$ abbreviates $\longrightarrow\!\!\!\!\!\rightarrow_{\beta\eta}^{+}$
- $=^{+}$ abbreviates $=_{\beta\eta}^{+}$

The notion of $\beta\eta$-equality has a finitary quality to it: If for some $M, N \in \Lambda$, we have $M = N$, then there exists a term $R \in \Lambda$, such that $M \longrightarrow\!\!\!\!\!\rightarrow R$ and $N \longrightarrow\!\!\!\!\!\rightarrow R$. But the $\longrightarrow\!\!\!\!\!\rightarrow$ relation has a finitary quality to it: If $A \longrightarrow\!\!\!\!\!\rightarrow B$, then there exists $n \in \mathbb{Z}^{+}$, such that $A(\longrightarrow)^{n}B$, or put another way, $A \longrightarrow B_1 \longrightarrow \cdots \longrightarrow B_{n-1} \longrightarrow B$, for some $\vec{B} \in \Lambda$. Hence if $M \longrightarrow\!\!\!\!\!\rightarrow R$ and $N \longrightarrow\!\!\!\!\!\rightarrow R$, there exists $m, n \in \mathbb{Z}^{+}$, such that M and N $\beta\eta$-reduce to R in m and n single-step $\beta\eta$-reductions, respectively.

3.6 Abbreviation: f^k. Through an abuse of notation we abbreviate $\lambda\nu.(f^k\ \nu)$ by f^k. This may look like a one-step η-reduction, but it is not.

3.7 Definition: *The $\lambda\mathbf{K}$-calculus, the $\lambda\mathbf{I}$-calculus.* The $\lambda\mathbf{K}$-calculus is a theory that studies the behaviour of $\Lambda_{\mathbf{K}}$ under the rule $\boldsymbol{\beta\eta}$ (it is also known as the $\lambda\mathbf{K}\beta\eta$-calculus). The $\lambda\mathbf{I}$-calculus is a theory that studies the behaviour of $\Lambda_{\mathbf{I}}$ under the rule $\boldsymbol{\beta\eta}$ (it is also known as the $\lambda\mathbf{I}\beta\eta$-calculus).

3.8 Definition: *The Axioms of the λ-Calculus* (see [1, Definition 2.1.4, Chapter 2, Page 23]). The λ-calculus is axiomatised by the following axioms and rules:

i. β-conversion: $((\lambda x.M)\ N) = M[x := N]$

ii. $M = M$

iii. If $M = N$, then $N = M$

iv. If $M = N$, and $N = P$, then $M = P$

v. If $M = N$, then $(M\ P) = (N\ P)$

vi. If $M = N$, then $(P\ M) = (P\ N)$

vii. ξ-rule: If $M = N$, $\lambda x.M = \lambda x.N$

for all $M, N, P \in \Lambda$.

Throughout this work, wherever we refer to the "λ-calculus," we intend, by default, the $\lambda\mathbf{K}$-calculus.

Many variants of these calculi exist today. They differ in the addition or removal of reduction rules, and the possible imposition of a system of types on the λ-terms.

3.9 DEFINITION: *Head Normal Form.* A λ-term M is *a head normal form* if $M = \lambda\vec{x}.(y\ \vec{N})$ for $\vec{x}, y \in \mathsf{Symbols}$, and $\vec{N} \in \Lambda$.

3.10 THEOREM: (Church-Rosser)

- $\boldsymbol{\beta}, \boldsymbol{\beta\eta}$ *are CR.*
- For $M, N \in \Lambda$, if $M =_\beta N$, then $M \longrightarrow\!\!\!\!\!\rightarrow_\beta P$, and $N \longrightarrow\!\!\!\!\!\rightarrow_\beta P$, for some $P \in \Lambda$.
- For $M, N \in \Lambda$, if $M =_{\beta\eta} N$, then $M \longrightarrow\!\!\!\!\!\rightarrow_{\beta\eta} P$, and $N \longrightarrow\!\!\!\!\!\rightarrow_{\beta\eta} P$, for some $P \in \Lambda$.

Proof: See Theorem 3.2.8, and Theorem 3.3.9, in Barendregt's text on the λ-calculus [1]. ■

3.11 THEOREM: *If $M, N \in \Lambda$ have* R-nf *M', N', respectively, and if $M =_{\mathsf{R}} N$, then $M' =_\alpha N'$.*

Proof: Since $=_{\mathsf{R}}$ is an equivalence relation, it is reflexive, symmetric, and transitive. It follows that $N' =_{\mathsf{R}} N =_{\mathsf{R}} M =_{\mathsf{R}} M'$. We want to show that if $N' =_{\mathsf{R}} M'$, for M', N' that are R-nf, then $N' =_\alpha M'$. Since M', N' are R-nf, they contain no R-redexes. It follows that they must be equal by reflexivity, which is taken modulo α. Our claim follows. ■

3.12 DEFINITION: *Solvability* [1, Definition 2.2.10, Page 41]. A λ-term M is said to be *solvable* if there exist for some $k \in \mathbb{N}$, k λ-terms $\vec{N}$, such that

$$(M\ \vec{N}) \quad \longrightarrow\!\!\!\!\!\rightarrow_{\mathsf{R}} \quad \mathbf{I} \tag{23}$$

If there exist such $\vec{N} \in \Lambda_{\mathbf{I}}$, M is said to be $\lambda\mathbf{I}$-solvable. If there exist such $\vec{N} \in \Lambda_{\mathbf{K}}$, M is said to be $\lambda\mathbf{K}$-solvable.

The solvability properties of a term are important in understanding how this λ-term behaves when applied to some arguments, that is, how these arguments are *used* by the given λ-term. Terms that are not solvable are considered *meaningless* [1, Section 2§2, Page 41–42].

Heavy use is made of solvability in the $\lambda\mathbf{I}$-calculus, where the variable of an abstraction must occur free in the body of the abstraction, i.e., be used. Whereas in the $\lambda\mathbf{K}$-calculus, we can make use of the $\mathbf{K}$ combinator (and other expressions in $\Lambda_{\mathbf{K}} - \Lambda_{\mathbf{I}}$) in order to "abandon" unwanted λ-terms, in the $\lambda\mathbf{I}$-calculus, on the other hand, those terms must be "used up". Solvability provides a way of "using up" a λ-term.

3.13 DEFINITION: *Böhm Trees*[3] [1, Definition 10.1.3, Page 216]. A *Böhm tree* is a [possibly infinite] tree associated with a λ-term. The Böhm tree of $M \in \Lambda$ is denoted by $\mathsf{BT}(M)$, and is defined inductively as:

$$\mathsf{BT}(M) = \bot \tag{24}$$

if M is unsolvable (i.e., a single node labelled $\bot$.)

$$\tag{25}$$

For any λ-term M, $\mathsf{BT}(M)$ is finite if and only if M has a normal form. In connection with this, Barendregt makes the beautiful observation that terms relate to their Böhm trees in the same way that the real numbers relate to their continued fractions [1, Section 10§1 – 10§2, Page 215–245]. Some relations between λ-terms and real numbers:

[3] Böhm trees are named after the logician *Corrado Böhm* [1923 –].

- Continued fractions are [possibly infinite] representations of real numbers, just as Böhm trees are [possible infinite] representations of λ-terms.
- Rational numbers and normalisable λ-terms are represented by finite continued fractions and finite Böhm trees, respectively.
- Irrational numbers and non-normalisable λ-terms are represented by infinite continued fractions and infinite Böhm trees, respectively.
- Endowed with the appropriate topology, normalisable λ-terms form a *dense* subset of Λ, in pretty much the same way the set $\mathbb{Q}$ of rational numbers forms a dense subset of the set $\mathbb{R}$ of real numbers.

4 Lamda Definability

4.1 Preliminaries

λ-Definability concerns itself with representing logical and mathematical entities in the λ-calculus: λ-terms are used to represent numbers, ordered n-tuples, boolean values, functions, and other structures.

We begin our excursion into λ-definability with the construction of a system of numerals, and the elementary number-theoretic functions defined over it. The numerals we consider here, and which are the default numerals used throughout this entire work, are known as the *Church numerals*, named after the logician *Alonzo Church* [1903 – 1995], who was the first to define and then make use of them.

4.2 Church Numerals

Suppose we have some λ-expression representing the number 0, and a λ-expression s representing the successor function. Then $(s\ z)$ represents the number 1, $(s\ (s\ z))$ represents the number 2, etc. The n-th Church numeral is thus defined by abstracting s and z over the n-th application of s to z:

4.1 DEFINITION: *Church numerals.* The n-th Church numeral, denoted by $\ulcorner n \urcorner$ is defined as

$$\begin{aligned} \ulcorner 0 \urcorner &= \lambda sz.z && (26)\\ \ulcorner n+1 \urcorner &= \lambda sz.\underbrace{(s\ \cdots(s}_{n+1}\ z)\cdots) \\ &= \lambda sz.(s^{n+1}\ z) && (27) \end{aligned}$$

Having defined Church numerals, let us define the elementary number-theoretic functions over them, starting with the successor function.

We begin by noticing that since a Church numeral is an abstraction over the n-th application of some successor function to some zero numeral, it is a simple matter to convert between Church numerals and any other kind of numerals, given a representation z' for zero in the alternate system, and an appropriate successor function s':

$$(\ulcorner n \urcorner\ s'\ z') \longrightarrow\!\!\!\!\!\rightarrow \underbrace{(s'\cdots(s'}_{n}\ z')\cdots) \tag{28}$$

Applying s' to both sides of (28), we get:

$$(s'\ (\ulcorner n \urcorner\ s'\ z')) \longrightarrow\!\!\!\!\!\rightarrow \underbrace{(s'\cdots(s'}_{n+1}\ z')\cdots) \tag{29}$$

We abstract the variables s and z for s' and z' over the left-hand side in (28), to get a numeral in Church's system:

$$\ulcorner n+1 \urcorner \quad = \quad \lambda sz.(s\ (\ulcorner n \urcorner\ s\ z)) \tag{30}$$

Since a λ-term which computes the successor function maps $\ulcorner n \urcorner$ to $\ulcorner n+1 \urcorner$, we obtain such a λ-term by abstracting n for $\ulcorner n \urcorner$ over the right-hand side of (30):

4.2 PROPOSITION: *The expression*

$$\mathbf{Succ}_{\mathsf{Church}} \quad = \quad \lambda nsz.(s\ (n\ s\ z)) \tag{31}$$

computes the successor function on Church numerals.

Proof: Verify that $(\mathbf{Succ}_{\mathsf{Church}}\ \ulcorner n \urcorner) \longrightarrow\!\!\!\!\!\rightarrow \ulcorner n+1 \urcorner$. ■

We stated that a Church numeral is an abstraction of a successor and a zero over the n-th application of the given successor to the given zero. This observation served as a useful metaphor in deriving a successor function over the Church numerals. We now offer the more general observation that Church numerals are abstractions over the n-th composition of some function: Given a function f, the n-th composition of f is

$$f^n \quad = \quad \lambda x.\underbrace{(f \cdots (f}_{n}\ x) \cdots) \tag{32}$$

The 0-th composition of f is defined to be the identity combinator $\mathbf{I} = \lambda x.x$. Abstracting f over f^n is α-equivalent to the n-th Church numeral. We thus have a convenient way to compute the bounded composition of a λ-term f since

$$(\ulcorner n \urcorner\ f) \quad \longrightarrow\!\!\!\!\!\rightarrow \quad f^n \tag{33}$$

We will be using (33) implicitly, but extensively, throughout this work.

In particular, we now make use of the above property in order to derive a λ-term which computes the addition function on Church numerals: The λ-term $(\ulcorner a \urcorner\ \mathbf{Succ}_{\mathsf{Church}})$ computes the function that adds $\ulcorner a \urcorner$ to any Church numeral. Therefore

$$(\ulcorner a \urcorner\ \mathbf{Succ}_{\mathsf{Church}}\ \ulcorner b \urcorner) \quad \longrightarrow\!\!\!\!\!\rightarrow \quad \ulcorner a+b \urcorner \tag{34}$$

A λ-term that computes the addition function is obtained by abstracting a and b for $\ulcorner a \urcorner$ and $\ulcorner b \urcorner$ respectively, over the left-hand side of (34):

4.3 PROPOSITION: *The expression*

$$\mathbf{Add}_{\mathsf{Church}} \quad = \quad \lambda ab.(a\ \mathbf{Succ}_{\mathsf{Church}}\ b) \tag{35}$$

computes the addition function on Church numerals.

Proof: Verify that

$$(\mathbf{Add}_{\mathsf{Church}}\ \ulcorner a \urcorner\ \ulcorner b \urcorner) \quad \longrightarrow\!\!\!\!\!\rightarrow \quad \ulcorner a+b \urcorner \tag{36}$$

■

Similarly, $(\mathbf{Add}_{\mathsf{Church}}\ \ulcorner a \urcorner)$ computes the function that adds $\ulcorner a \urcorner$ to a Church numeral, and $(\ulcorner b \urcorner\ (\mathbf{Add}_{\mathsf{Church}}\ \ulcorner a \urcorner))$ computes the b-th composition of that function, which is the function that adds $\ulcorner a \times b \urcorner$ to a Church numeral. Therefore

$$(\ulcorner b \urcorner\ (\mathbf{Add}_{\mathsf{Church}}\ \ulcorner a \urcorner)\ \ulcorner 0 \urcorner) \quad \longrightarrow\!\!\!\!\!\rightarrow \quad \ulcorner a \times b \urcorner \tag{37}$$

A λ-term that computes the multiplication function on Church numerals is obtained by abstracting the variables a and b for $\ulcorner a \urcorner$ and $\ulcorner b \urcorner$ respectively over the left-hand side of (37):

4.4 PROPOSITION: *The expression*

$$\mathbf{Times}_{\mathsf{Church}} \quad = \quad \lambda ab.(b\ (\mathbf{Add}_{\mathsf{Church}}\ a)\ \ulcorner 0 \urcorner) \tag{38}$$

computes the multiplication function on Church numerals.

Proof: Verify that $(\mathbf{Times}_{\mathsf{Church}} \ulcorner a \urcorner \ulcorner b \urcorner) \twoheadrightarrow \ulcorner a \times b \urcorner$. ■

It should be clear by now how to construct the exponentiation function:

4.5 PROPOSITION: *The expression*

$$\mathbf{Power}_{\mathsf{Church}} = \lambda ab.(b\ (\mathbf{Times}_{\mathsf{Church}}\ a)\ \ulcorner 1 \urcorner) \tag{39}$$

computes the exponentiation function on Church numerals.

Proof: Verify that $(\mathbf{Power}_{\mathsf{Church}} \ulcorner a \urcorner \ulcorner b \urcorner) \twoheadrightarrow \ulcorner a^b \urcorner$. ■

Concerning exponentiation, however, Church [4, Chapter II, Page 10] noticed that

$$(\ulcorner b \urcorner \ulcorner a \urcorner) \longrightarrow \ulcorner a^b \urcorner \tag{40}$$

and so exponentiation can also be defined as $\lambda ab.(b\ a)$. This fact, which is easy to verify, is suggested by the similarity between the algebraic laws of exponentiation and composition.

We can further iterate over the exponentiation function to obtain even faster growing functions, and arrive, in fact, at *Ackermann's function*:

4.6 DEFINITION: *Ackermann's Function*[4]. Ackermann's function $\mathfrak{a}$ is defined inductively as follows:

$$\mathfrak{a}(0, q) = q + 1 \tag{41}$$
$$\mathfrak{a}(p + 1, 0) = \mathfrak{a}(p, 1) \tag{42}$$
$$\mathfrak{a}(p + 1, q + 1) = \mathfrak{a}(p, \mathfrak{a}(p + 1, q)) \tag{43}$$

The expansion of $\mathfrak{a}(p+1, q)$, after q applications of (43), followed by a single application of (42), is

$$\mathfrak{a}(p + 1, q) = \underbrace{\mathfrak{a}(p, \cdots \mathfrak{a}(p + 1, 0) \cdots)}_{q+1} \tag{44}$$
$$= \underbrace{\mathfrak{a}(p, \cdots \mathfrak{a}(p, 1) \cdots)}_{q+1} \tag{45}$$

We curry over p, so $\mathfrak{a}_p(q) = \mathfrak{a}(p, q)$. We now further rewrite (45) as

$$\mathfrak{a}_{p+1}(q) = \underbrace{\mathfrak{a}_p(\cdots \mathfrak{a}_p(1) \cdots)}_{q+1} \tag{46}$$

Let A_p be a λ-term computing $\mathfrak{a}_p$ on Church numerals. We compose A_p using Church numerals, exploiting the following property: For any λ-term f, and any $n \in \mathbb{Z}^+$ we have

$$(\ulcorner n \urcorner\ f) = f^n \tag{47}$$

Thus, equation (46) can be rewritten in the λ-calculus as

$$A_{p+1} = \underbrace{A_p(\cdots A_p(\ulcorner 1 \urcorner) \cdots)}_{q+1} \tag{48}$$
$$= (\ulcorner q + 1 \urcorner\ A_p\ \ulcorner 1 \urcorner) \tag{49}$$
$$= (\mathbf{Succ}_{\mathsf{Church}}\ \ulcorner q \urcorner\ A_p\ \ulcorner 1 \urcorner) \tag{50}$$

The function $f : A_p \to A_{p+1}$ is λ-definable by abstracting the variable q for $\ulcorner q \urcorner$ over A_p in (50):

$$f = \lambda a_p q.(\mathbf{Succ}_{\mathsf{Church}}\ q\ a_p\ \ulcorner 1 \urcorner) \tag{51}$$

[4] Wilhelm Ackermann, [1896 – 1962]

To compute $\mathfrak{a}(p,q) = \mathfrak{a}_p(q)$ we need to apply the p-th composition of f to A_0 (which computes $\mathfrak{a}_0$ on Church numerals). As can be seen from (41), A_0 is just $\mathbf{Succ}_{\mathsf{Church}}$. Abstracting over p and q yields:

$$\begin{aligned}
\mathbf{Ack}_{\mathsf{Church}} &= \lambda pq.(p\ f\ A_0\ q) && (52)\\
&=_\eta \lambda p.(p\ f\ A_0) && (53)\\
&= \lambda p.(p\ (\lambda aq.(\mathbf{Succ}_{\mathsf{Church}}\ q\ a\ \ulcorner 1\urcorner))\ \mathbf{Succ}_{\mathsf{Church}}) && (54)
\end{aligned}$$

4.7 Proposition: *The expression*

$$\mathbf{Ack}_{\mathsf{Church}} = \lambda p.(p\ (\lambda aq.(a\ (q\ a\ \ulcorner 1\urcorner)))\ \mathbf{Succ}_{\mathsf{Church}}) \quad (55)$$

computes Ackermann's function on Church numerals.

Proof: It is straightforward to verify that $\mathbf{Ack}_{\mathsf{Church}}$ satisfies the following three equations:

$$\begin{aligned}
(\mathbf{Ack}_{\mathsf{Church}}\ \ulcorner 0\urcorner\ \ulcorner q\urcorner) &= \ulcorner q+1\urcorner && (56)\\
(\mathbf{Ack}_{\mathsf{Church}}\ \ulcorner p+1\urcorner\ \ulcorner 0\urcorner) &= (\mathbf{Ack}_{\mathsf{Church}}\ \ulcorner p\urcorner\ \ulcorner 1\urcorner) && (57)\\
(\mathbf{Ack}_{\mathsf{Church}}\ \ulcorner p+1\urcorner\ \ulcorner q+1\urcorner) &= (\mathbf{Ack}_{\mathsf{Church}}\ \ulcorner p\urcorner\ (\mathbf{Ack}_{\mathsf{Church}}\ \ulcorner p+1\urcorner\ \ulcorner q\urcorner)) && (58)
\end{aligned}$$

which correspond to (41)–(43) respectively. ■

Our ability to construct Ackermann's the way we did might come as a surprise: We used Church numerals as a bounded iteration mechanism, and this was the only iteration mechanism we needed. However, Ackermann's function is not primitive recursive, and bounded iteration is the only form of iteration used to generate the primitive recursive functions. Furthermore, if we were to add unbounded iteration to the definition of the primitive recursive functions, we would have general recursion, and could definitely compute Ackermann's function. So it would seem as if bounded iteration is not powerful enough in order to define Ackermann's function. In order to understand how we could define Ackermann the way we did, we need to look at what we are applying bounded iteration to. We use Church numerals as an iteration mechanism twice: First, to iterate over a_p, and second, to iterate over f (see (51) and (55)). The function a_p maps integers to integers, and the function f is a higher-order function, mapping functions from integers to integers to functions from integers to integers. A careful examination of the primitive recursion schema reveals that bounded iteration is applied to first-order functions. So as we have seen, if we relax this restriction, we can use bounded iteration to generate functions that are not primitive recursive.

In his book *Proofs and Types* [7, Section 7.3.2, Page 51], Girard mentions that Ackermann's function is definable through finite iteration over some "reasonable" function of a complex type. Our construction provides such a reasonable function in the pure λ-calculus.

4.3 Boolean Values and Conditional Statements

We would like to have Boolean values, Boolean functions and predicates. The purpose for defining Boolean values is to have a selection mechanism of the form

$$\mathsf{if}\ \textit{condition}\ \mathsf{then}\ \textit{do-if-true}\ \mathsf{else}\ \textit{do-if-false}\ \mathsf{fi}$$

Since the purpose of introducing Boolean values is to be able to use them to select between two possibilities (i.e., expressions), we build the selection mechanism into the Boolean values themselves, thus we define $\mathbf{T}$, the Boolean value *true,* as taking two possibilities and selecting (i.e., evaluating to) the first; similarly, we define $\mathbf{F}$, the Boolean value *false,* as taking two possibilities and selecting the second.

4.8 Definition: *Boolean values.*

$$\begin{aligned}
\mathbf{T} &= \lambda xy.x && (59)\\
\mathbf{F} &= \lambda xy.y && (60)
\end{aligned}$$

It is now a simple matter to define the standard Boolean operators: The combinator **Not** which takes an argument x and returns **F** if x is true, and **T** otherwise; The combinator **And** takes two arguments and returns **T** if both are true, **F** otherwise; And the Boolean combinator **Or** takes two arguments and returns **T** if either is true, **F** otherwise. Notice that there is no need to define λ-expressions for if, then, else, or fi, since the Boolean value itself performs the selection.

4.9 PROPOSITION: *The following expressions correspond respectively to the Boolean functions* $\neg$, $\wedge$ *and* $\vee$*:*

$$\begin{aligned} \mathbf{Not} &= \lambda x.(x\ \mathbf{F}\ \mathbf{T}) && (61)\\ \mathbf{And} &= \lambda xy.(x\ (y\ \mathbf{T}\ \mathbf{F})\ \mathbf{F}) && (62)\\ \mathbf{Or} &= \lambda xy.(x\ \mathbf{T}\ (y\ \mathbf{T}\ \mathbf{F})) && (63) \end{aligned}$$

Note that it is straightforward to verify [by enumeration] that for any boolean value b, we have

$$(b\ \mathbf{T}\ \mathbf{F}) \quad \twoheadrightarrow \quad b \tag{64}$$

and so the definitions of **And** (in (62)) and **Or** (in (63)), can be simplified. We refrain from this simplification because the current definitions of **And** and **Or** can be read off directly from the *truth tables* for the respective boolean functions, and in that sense they are more intuitive.

Proof: Verify that these Boolean functions behave as the corresponding truth tables suggest. ■

Boolean values would be of little use to us without predicates. Since we have just defined a system of numerals, it would seem reasonable to define the $\mathbf{Zero?}_{\mathsf{Church}}$ predicate. Church numerals, which are the arguments to the $\mathbf{Zero?}_{\mathsf{Church}}$ predicate take two arguments and apply the first to the second some number of times. Our approach to defining $\mathbf{Zero?}_{\mathsf{Church}}$ would therefore be to define two expressions A and B such that for all $n \in \mathbb{N}$, we have

$$\begin{aligned} (\ulcorner 0 \urcorner\ A\ B) &\twoheadrightarrow \mathbf{T} && (65)\\ (\ulcorner n+1 \urcorner\ A\ B) &\twoheadrightarrow \mathbf{F} && (66) \end{aligned}$$

Clearly

$$\begin{aligned} (\ulcorner 0 \urcorner\ A\ B) &= ((\lambda ab.b)\ A\ B) && (67)\\ &\twoheadrightarrow B && (68)\\ &= \mathbf{T} && (69) \end{aligned}$$

So $B = \mathbf{T}$. Solving for A, we have $\ulcorner n+1 \urcorner = \lambda ab.(a\ M_{a,b})$ where $M_{a,b}$ is one of b, $(a\ b)$, $(a\ (a\ b))$, etc., depending on n. So for $n > 0$ we apply b [at least once] while for $n = 0$ we completely ignore b. Regardless of what b is applied to, we would like the application to evaluate to **F**. So we let $A = \lambda x.\mathbf{F}$. We now define $\mathbf{Zero?}_{\mathsf{Church}}$ to take a Church numeral n and apply it to the above expressions A and B. Thus:

4.10 PROPOSITION: *The expression*

$$\mathbf{Zero?}_{\mathsf{Church}} \quad = \quad \lambda n.(n\ (\lambda x.\mathbf{F})\ \mathbf{T}) \tag{70}$$

computes the zero predicate on Church numerals.

Proof: Verify that for any $n \in \mathbb{N}$, we have

$$\begin{aligned} (\mathbf{Zero?}_{\mathsf{Church}}\ \ulcorner 0 \urcorner) &\twoheadrightarrow \mathbf{T} && (71)\\ (\mathbf{Zero?}_{\mathsf{Church}}\ \ulcorner n+1 \urcorner) &\twoheadrightarrow \mathbf{F} && (72) \end{aligned}$$

■

4.4 Aggregate Data Structures

Numerals and Booleans are common data types in many programming languages, and defining them in the λ-calculus gives λ-definability the expressibility of a programming language. Before we proceed any further, we would like to incorporate one additional concept from programming languages, that of *aggregate data structures.* The most convenient data structure for our purposes is the *ordered n-tuple.*

The encoding $[x_1, \ldots, x_n]$ of the ordered n-tuple $\langle x_1, \ldots, x_n \rangle$ must allow us to project along any dimension $k \in \{1, \ldots, n\}$, and therefore $[x_1, \ldots, x_n]$ can be defined to take a *selector* of the form $(\lambda x_1 \cdots x_n . x_k)$ and pass onto it $x_1, \ldots, x_n$. Thus

$$[x_1, \ldots, x_n] \quad = \quad \lambda s.(s\ x_1 \cdots x_n) \tag{73}$$

The k-th projection of an n-tuple can be defined to take an ordered n-tuple and pass onto it the k-th *selector:*

$$\pi_k^n \quad = \quad \lambda t.(t\ (\lambda x_1 \cdots x_n . x_k)) \tag{74}$$

Finally, the $n+1$-tuple constructor can be defined by abstracting $x_1, \ldots, x_{n+1}$ over $[x_1, \ldots, x_{n+1}]$ to give:

4.11 PROPOSITION: *The following expressions are an $n+1$-tuple constructor, and a k-th n-tuple projection function:*

$$\underbrace{,[.],}_{n+1} = \lambda x_1 \cdots x_n . [x_1, \ldots, x_{n+1}] \tag{75}$$

$$= \lambda x_1 \cdots x_{n+1} s.(s\ x_1 \cdots x_{n+1}) \tag{76}$$

$$\pi_k^n = \lambda t.(t\ (\lambda x_1 \cdots x_n . x_k)) \tag{77}$$

Proof: Verify that

$$(\pi_k^{n+1}\ \underbrace{(,[.],}_{n+1} x_1 \cdots x_{n+1})) \quad \twoheadrightarrow \quad (\pi_k^{n+1}\ [x_1, \ldots, x_{n+1}]) \tag{78}$$

$$\twoheadrightarrow \quad x_k \tag{79}$$

■

One might wonder why we chose not to define the ordered n-tuple (for $n > 2$) as nested ordered pairs. The reason is that this definition would lead to anomalous solutions which do not exist given the representation in (73). Consider for example the following claim:

4.12 CLAIM: *Given the representation we chose for ordered n-tuples, there exists no expression M such that $[M, M] = [M, M, M]$.*

Proof: Assume there exists such M. Then applying π_3^3 to both sides of the equation we get: $(\pi_3^3\ [M, M, M]) = (\pi_3^3\ [M, M])$. But:

$$(\pi_3^3\ [M, M, M]) \quad \twoheadrightarrow \quad M. \tag{80}$$

$$(\pi_3^3\ [M, M]) \quad \twoheadrightarrow \quad ((\lambda x_1 x_2 x_3 . x_3)\ M\ M) \tag{81}$$

$$\twoheadrightarrow \quad \lambda x_3 . x_3 \equiv \mathbf{I}. \tag{82}$$

so $M = \mathbf{I}$. We now substitute $\mathbf{I}$ for M in the equation $[M, M] = [M, M, M]$ to obtain

$$[\mathbf{I}, \mathbf{I}] \quad = \quad [\mathbf{I}, \mathbf{I}, \mathbf{I}] \tag{83}$$

which is a contradiction, by Theorem 3.11, because both the right and the left-hand side of (83) have different normal forms. It follows that the equation $[M, M] = [M, M, M]$ has no solution. There is a solution however, given the representation of an ordered triple as a nested ordered pair: Let $M = (\Phi\ (\lambda m.[m, m]))$ (where Φ is some fixed-point combinator). Clearly $M = [M, M]$,

and therefore $[M, M] = [M, [M, M]]$. Since we found an equation for which the existence of a solution depends on the choice between two representations for ordered n-tuples, it follows that these representations are not equivalent. ■

Now that we have defined ordered n-tuples, we can return to λ-defining the elementary number-theoretic functions.

4.13 DEFINITION: *The Monus Function.* The monus function, denoted by $\dot{-}$ (written as a *dot* over a *minus* sign) is defined as follows:

$$a \dot{-} b \quad = \quad \begin{cases} a - b & \text{if } a \geq b \\ 0 & \text{otherwise} \end{cases} \tag{84}$$

The attentive reader may have noticed that we have yet to define a predecessor function, and may have wondered why we chose to postpone its construction. In fact, the predecessor function $\mathbf{Pred}_{\mathsf{Church}}$ over the Church numerals poses a special challenge: Church numerals, as stated before, are abstractions over the n-th composition of some function, so

$$(\mathbf{Pred}_{\mathsf{Church}} \ulcorner n+1 \urcorner) \tag{85}$$

$$= \quad (\mathbf{Pred}_{\mathsf{Church}} \ (\lambda f x. \underbrace{(f \cdots (f}_{n+1} x) \cdots)) \tag{86}$$

$$\longrightarrow\!\!\!\!\!\rightarrow \quad \lambda f x. \underbrace{(f \cdots (f}_{n} x) \cdots) \tag{87}$$

It is hardly obvious how to "peal off" one application of f in $\ulcorner n+1 \urcorner$ to yield $\ulcorner n \urcorner$. The following idea is due to Kleene: Consider the function f mapping $[a, b]$ to $[(\mathbf{Succ}_{\mathsf{Church}} \ a), a]$. We can define f as follows:

$$\lambda p.([(\mathbf{Succ}_{\mathsf{Church}} \ (\pi_1^2 \ p)), (\pi_1^2 \ p)]) \tag{88}$$

Notice that for any $x \in \Lambda$

$$(\ulcorner n+1 \urcorner \ f \ [\ulcorner 0 \urcorner, x]) \quad \longrightarrow\!\!\!\!\!\rightarrow \quad (f^{n+1} \ [\ulcorner 0 \urcorner, x]) \tag{89}$$

$$\longrightarrow\!\!\!\!\!\rightarrow \quad [\ulcorner n+1 \urcorner, \ulcorner n \urcorner] \tag{90}$$

Although the predecessor function is undefined for zero, it would be useful to have

$$(\mathbf{Pred}_{\mathsf{Church}} \ \ulcorner 0 \urcorner) \quad \longrightarrow\!\!\!\!\!\rightarrow \quad \ulcorner 0 \urcorner \tag{91}$$

For example, it would make the definition of the *monus* function much simpler. And so we choose x to be $\ulcorner 0 \urcorner$. Hence

$$(\ulcorner n \urcorner \ f \ [\ulcorner 0 \urcorner, \ulcorner 0 \urcorner]) \quad \longrightarrow\!\!\!\!\!\rightarrow \quad [\ulcorner n \urcorner, \ulcorner n \dot{-} 1 \urcorner] \tag{92}$$

and so

$$(\pi_2^2 \ (\ulcorner n \urcorner \ f \ [\ulcorner 0 \urcorner, \ulcorner 0 \urcorner])) \quad \longrightarrow\!\!\!\!\!\rightarrow \quad \ulcorner n \dot{-} 1 \urcorner \tag{93}$$

The predecessor function is defined by abstracting the variable n for $\ulcorner n \urcorner$ over $(\pi_2^2 \ (\ulcorner n \urcorner \ f \ [\ulcorner 0 \urcorner, \ulcorner 0 \urcorner]))$:

4.14 PROPOSITION: *The λ-expression $\mathbf{Pred}_{\mathsf{Church}}$ given below computes the predecessor function on Church numerals:*

$$\mathbf{Pred}_{\mathsf{Church}} \quad = \quad \lambda n(\pi_2^2 \ (n \ (\lambda p.[(\mathbf{Succ}_{\mathsf{Church}} \ (\pi_1^2 \ p)), (\pi_1^2 \ p)]) \ [\ulcorner 0 \urcorner, \ulcorner 0 \urcorner])) \tag{94}$$

Proof: Verify that $(\mathbf{Pred}_{\mathsf{Church}} \ \ulcorner n \urcorner) \longrightarrow\!\!\!\!\!\rightarrow \ulcorner n \dot{-} 1 \urcorner$. ■

The same technique carries over to other number-theoretic functions. For example, consider the function f mapping $[a, b]$ to $[(\mathbf{Succ}_{\mathsf{Church}}\ a), (\mathbf{Times}_{\mathsf{Church}}\ a\ b)]$. It is a simple matter to define f:

$$\begin{array}{l}\lambda p.[(\mathbf{Succ}_{\mathsf{Church}}\ (\pi_1^2\ p)),\\ \quad (\mathbf{Times}_{\mathsf{Church}}\ (\pi_1^2\ p)\ (\pi_2^2\ p))]\end{array} \tag{95}$$

Finally, observe that

$$\begin{array}{rcll}(\ulcorner n \urcorner\ f\ [\ulcorner 1 \urcorner, \ulcorner 1 \urcorner]) & \longrightarrow\!\!\!\!\rightarrow & (f^n\ [\ulcorner 2 \urcorner, \ulcorner 1 \urcorner]) & (96)\\ & \longrightarrow\!\!\!\!\rightarrow & [\ulcorner n+1 \urcorner, \ulcorner n! \urcorner] & (97)\end{array}$$

So by taking the second tuple of the last expression, and abstracting the variable n for $\ulcorner n \urcorner$ over the left-hand side of (96), we get a definition for the factorial function:

$$\begin{array}{l}\mathbf{Fact}_{\mathsf{Church}}\\ \quad = \quad \lambda n.(\pi_2^2\ (n\ (\lambda p.[(\mathbf{Succ}_{\mathsf{Church}}\ (\pi_1^2\ p)),\\ \qquad\qquad\qquad (\mathbf{Times}_{\mathsf{Church}}\ (\pi_1^2\ p)\ (\pi_2^2\ p))])\\ \qquad\qquad [\ulcorner 1 \urcorner, \ulcorner 1 \urcorner]))\end{array} \tag{98}$$

Given the definition of the predecessor function, we can now λ-define the *monus* function over Church numerals (Definition 4.13 defined the monus function over $\mathbb{Z}^+$). The reason for λ-defining the monus function, rather than the minus function, has to do with the fact that the combinator we defined for the predecessor function satisfies (91): We need only apply the b-th composition of $\mathbf{Pred}_{\mathsf{Church}}$ to $\ulcorner a \urcorner$ to get $\ulcorner a \dot{-} b \urcorner$:

4.15 PROPOSITION: *The expression*

$$\mathbf{Monus}_{\mathsf{Church}} \quad = \quad \lambda ab.(b\ \mathbf{Pred}_{\mathsf{Church}}\ a) \tag{99}$$

computes the monus *function on Church numerals.*

Proof: Verify that $(\mathbf{Monus}_{\mathsf{Church}}\ \ulcorner a \urcorner\ \ulcorner b \urcorner) \longrightarrow\!\!\!\!\rightarrow \ulcorner a \dot{-} b \urcorner$. ■

Equality of Church numerals is defined in terms of $\mathbf{Monus}_{\mathsf{Church}}$:

4.16 PROPOSITION: *The expression* $\mathbf{Equal?}_{\mathsf{Church}}$ *defined below computes the equality predicate on Church numerals.*

$$\begin{array}{l}\mathbf{Equal?}_{\mathsf{Church}}\\ \quad = \quad \lambda ab.(\mathbf{And}\ (\mathbf{Zero?}_{\mathsf{Church}}\ (\mathbf{Monus}_{\mathsf{Church}}\ a\ b))\\ \qquad\qquad\qquad (\mathbf{Zero?}_{\mathsf{Church}}\ (\mathbf{Monus}_{\mathsf{Church}}\ b\ a)))\end{array} \tag{100}$$

Proof: Verify that for all $a, b \in \mathbb{N}, a \neq b$, we have

$$\begin{array}{rcll}(\mathbf{Equal?}_{\mathsf{Church}}\ \ulcorner a \urcorner\ \ulcorner a \urcorner) & \longrightarrow\!\!\!\!\rightarrow & \mathbf{T} & (101)\\ (\mathbf{Equal?}_{\mathsf{Church}}\ \ulcorner a \urcorner\ \ulcorner b \urcorner) & \longrightarrow\!\!\!\!\rightarrow & \mathbf{F} & (102)\end{array}$$

■

The next proposition is similar, and is given without proof.

4.17 PROPOSITION: *The expressions below compute the functions* $\neq, \leq, \geq, <, >$ *on Church numerals respectively:*

$$\begin{array}{l}\mathbf{NotEqual?}_{\mathsf{Church}}\\ \quad = \quad \lambda ab.(\mathbf{Not}\ (\mathbf{Equal?}_{\mathsf{Church}}\ a\ b)) \qquad (103)\\ \mathbf{LessThanOrEqual?}_{\mathsf{Church}}\\ \quad = \quad \lambda ab.(\mathbf{Zero?}_{\mathsf{Church}}\ (\mathbf{Monus}_{\mathsf{Church}}\ a\ b)) \qquad (104)\end{array}$$

$$\begin{aligned}
&\mathbf{GreaterThanOrEqual?}_{\mathsf{Church}} \\
&\quad = \quad \lambda ab.(\mathbf{Zero?}_{\mathsf{Church}}\ (\mathbf{Monus}_{\mathsf{Church}}\ b\ a)) && (105)\\
&\mathbf{LessThan?}_{\mathsf{Church}} \\
&\quad = \quad \lambda ab.(\mathbf{Not}\ (\mathbf{GreaterThanOrEqual?}_{\mathsf{Church}}\ a\ b)) && (106)\\
&\mathbf{GreaterThan?}_{\mathsf{Church}} \\
&\quad = \quad \lambda ab.(\mathbf{Not}\ (\mathbf{LessThanOrEqual?}_{\mathsf{Church}}\ a\ b)) && (107)
\end{aligned}$$

5 Fixed Points

5.1 Single Fixed Points

In mathematics, a fixed point of a given function f is a point $x_0 \in \mathsf{Domain}(f) \cap \mathsf{Range}(f)$, that satisfies $f(x_0) = x_0$. If we consider the set of functions defined over the real line $\{f : \mathbb{R} \to \mathbb{R}\}$, we can find functions with finitely many, countably many, uncountably many, or no fixed points.

In the λ-calculus, the situation is somewhat similar:

5.1 DEFINITION: *A Fixed Point.* The *fixed-point* of a λ-term M is a λ-term x that satisfies

$$(M\ x) \quad = \quad x \tag{108}$$

A difference worth noting between the notion of a fixed point in mathematics, and the corresponding notion in the λ-calculus is that in the λ-calculus both M and x are λ-terms. By Intuition 2.2, both M and x can be thought of as functions. But what does it mean for a fixed point to be a function? We explore this question through the next example.

5.2 EXAMPLE: *The Factorial Function as a Fixed Point.* The partial function $\mathbf{F}_n$ maps $\ulcorner k \urcorner$ to $\ulcorner k! \urcorner$, for all $k = 1, \ldots, n$, and is undefined for all $k > n$. The higher-order function $g : \mathbf{F}_k \to \mathbf{F}_{k+1}$ for any $k \in \mathbb{N}$ is λ-definable as follows:

$$\begin{aligned}
g \quad = \quad & \lambda fk.(\mathbf{Zero?}_{\mathsf{Church}}\ k\ \ulcorner 1 \urcorner \\
& \qquad (\mathbf{Times}_{\mathsf{Church}}\ n\ (f\ (\mathbf{Pred}_{\mathsf{Church}}\ n)))) && (109)
\end{aligned}$$

What are the fixed points of g ? It can be seen by induction, that any fixed-point of g will map $\ulcorner n \urcorner \to \ulcorner n! \urcorner$ for all $n \in \mathbb{N}$, and so all fixed-points of g are extensions of the factorial function on Church numerals. In this sense, the factorial function on Church numerals can be considered to be the *least fixed point* of f.[5]

5.3 DEFINITION: *A Fixed-Point Combinator.* A combinator Φ is a *fixed-point combinator*, if for any $M \in \Lambda$, we have

$$(\Phi\ M) \quad = \quad (M\ (\Phi\ M)) \tag{110}$$

So $(\Phi\ M)$ is a fixed point of M.

5.4 DEFINITION: *Curry's Fixed-Point Combinator, Turing's Fixed-Point Combinator.*[6] The following are two well-known fixed-point combinators. *Curry's* fixed-point combinator, $\mathbf{Y}_{\mathsf{Curry}}$, is given by

$$\begin{aligned}
\mathbf{Y}_{\mathsf{Curry}} \quad = \quad & \lambda f.((\lambda x.(f\ (x\ x))) \\
& \qquad (\lambda x.(f\ (x\ x)))) && (111)
\end{aligned}$$

Turing's fixed-point combinator, $\mathbf{Y}_{\mathsf{Turing}}$ is given by

$$\begin{aligned}
\mathbf{Y}_{\mathsf{Turing}} \quad = \quad & ((\lambda xf.(f\ (x\ x\ f))) \\
& \qquad (\lambda xf.(f\ (x\ x\ f)))) && (112)
\end{aligned}$$

[5] For a more formal treatment of the ordering of functions, the reader is referred to any of the standard texts on lattices and domains [8].

[6] These terms are named after *Haskell Brooks Curry* [1900 – 1982], and *Alan Mathison Turing* [1912 – 1954].

Turing's fixed-point combinator satisfies for all $M \in \Lambda$:

$$(\mathbf{Y}_{\mathsf{Turing}}\ M) \quad \longrightarrow\!\!\!\!\!\rightarrow \quad (M\ (\mathbf{Y}_{\mathsf{Turing}}\ M)) \tag{113}$$

This property is a stronger property than (110) (Recall Observation 3.3).

Returning to Example 5.2, where the factorial function is defined as the fixed point of the higher-order function g, we are now in the position to provide an alternate definition for $\mathbf{Fact}_{\mathsf{Church}}$, using $\mathbf{Y}_{\mathsf{Curry}}$ and $\mathbf{Y}_{\mathsf{Turing}}$. We state the following claim without proof:

5.5 Claim: *Let g be defined as in* (109). *The combinators $\mathbf{Fact}'_{\mathsf{Church}}$ and $\mathbf{Fact}''_{\mathsf{Church}}$, defined below, both compute the factorial function on Church numerals:*

$$\mathbf{Fact}'_{\mathsf{Church}} = (\mathbf{Y}_{\mathsf{Curry}}\ g) \tag{114}$$

$$\mathbf{Fact}''_{\mathsf{Church}} = (\mathbf{Y}_{\mathsf{Turing}}\ g) \tag{115}$$

Another approach to defining recursive functions is to use the minimalisation operator (aka: "μ operator"):

5.6 Definition: *The μ Operator.* The μ operator is a partial map that takes a predicate P, defined over $\mathbb{Z}^+$, and returns the smallest number that satisfies P (if such a number exists).

5.7 Theorem: *The μ Operator is λ-Definable.*

Proof: Let Φ be any fixed-point combinator. We define

$$R_p = \lambda rn.(p\ n\ n\ (r\ (\mathbf{Succ}_{\mathsf{Church}}\ n))) \tag{116}$$

$$\mu = \lambda p.(\Phi\ R_p\ \ulcorner 0 \urcorner) \tag{117}$$

It is straightforward to verify that

$$(\Phi\ R_p\ \ulcorner n \urcorner) = \begin{cases} \ulcorner n \urcorner & \text{if } (p\ \ulcorner n \urcorner) \longrightarrow\!\!\!\!\!\rightarrow \mathbf{T} \\ (\Phi\ R_p\ \ulcorner n+1 \urcorner) & \text{otherwise} \end{cases}$$

■

5.2 Multiple Fixed Points

Multiple fixed points and multiple fixed-point combinators extend the notions of fixed points and fixed-point combinators, respectively. Just as we can define recursive functions by applying a fixed-point combinator to a higher-order function (such as g in (109)), we can define *mutually recursive* functions by applying a *multiple fixed-point combinator* to several higher-order functions.

5.8 Definition: *Multiple Fixed Points.* For all $n \in \mathbb{N}$, the *multiple fixed-points* of n λ-terms $A_1, \ldots, A_n$, are n λ-terms $x_1, \ldots, x_n$ that satisfy:

$$\begin{aligned} (A_1\ x_1 \cdots x_n) &= x_1 \\ &\cdots \\ (A_n\ x_1 \cdots x_n) &= x_n \end{aligned}$$

It is a simple matter to verify that when $n = 1$, this definition coincides with Definition 5.1.

5.9 Definition: *Multiple Fixed-Point Combinators.* The n λ-terms $\Phi_1, \ldots, \Phi_n$ are said to be fixed-point combinators for a set of n λ-terms, if for any n λ-terms $A_1, \ldots, A_n$, the λ-terms

$$\begin{aligned} &(\Phi_1\ A_1 \cdots A_n) \\ &\cdots \\ &(\Phi_n\ A_1 \cdots A_n) \end{aligned}$$

are multiple fixed-points of $A_1, \ldots, A_n$, respectively.

The proof of the following theorem is a generalisation of a beautiful proof due to Smullyan[7] [2, Pages 334–335].

5.10 THEOREM: *There exist multiple fixed-point combinators for a set of n λ-terms.*

Proof: Pick $n \in \mathbb{N}$. Using an ordinary fixed-point combinator Φ as in Definition 5.3, we can define a combinator M that satisfies:

$$(M\ x_0\ x_1 \cdots x_n) \quad = \quad (x_0\ (M\ x_1\ x_1 \cdots x_n) \quad \cdots \quad (M\ x_n\ x_1 \cdots x_n)) \tag{118}$$

A definition for M could be

$$M \quad = \quad (\Phi\ (\lambda m x_0 x_1 \cdots x_n.(x_0\ (m\ x_1\ x_1 \cdots x_n) \quad \cdots \quad (m\ x_n\ x_1 \cdots x_n)))) \tag{119}$$

In terms of M we can define for $k = 1, \ldots, n$, the k-th fixed-point combinator for n λ-terms:

$$\Phi_k \quad = \quad \lambda x_1 \cdots x_n.(M\ x_k\ x_1 \cdots x_n) \tag{120}$$

It is straightforward to verify that $\Phi_1, \ldots, \Phi_n$ satisfy Definition 5.9. ■

In the above derivation of multiple fixed-point combinators, we make use of an ordinary fixed-point combinator (for example, $\mathbf{Y}_{\mathsf{Curry}}$, or $\mathbf{Y}_{\mathsf{Turing}}$). For an alternate derivation of multiple fixed points using a fixed-point combinator[8], see Bekič's Theorem in Winskel's *The Formal Semantics of Programming Languages* [9, Chapter 10, Section]. Bekič himself referred to the theorem as the *bisection lemma* [3, Page 39].

5.11 EXAMPLES: *Defining the* $\mathbf{Even?}_{\mathsf{Church}}$, *and* $\mathbf{Odd?}_{\mathsf{Church}}$ *combinators as Multiple Fixed Points.* Let

$$E \quad = \quad \lambda e o n.(\mathbf{Zero?}_{\mathsf{Church}}\ n\ \mathbf{T}\ (o\ (\mathbf{Pred}_{\mathsf{Church}}\ n))) \tag{121}$$

$$O \quad = \quad \lambda e o n.(\mathbf{Zero?}_{\mathsf{Church}}\ n\ \mathbf{F}\ (e\ (\mathbf{Pred}_{\mathsf{Church}}\ n))) \tag{122}$$

Let Φ_1, Φ_2 be multiple fixed-point combinators for two expressions. We can now define:

$$\mathbf{Even?}_{\mathsf{Church}} \quad = \quad (\Phi_1\ E\ O) \tag{123}$$

$$\mathbf{Odd?}_{\mathsf{Church}} \quad = \quad (\Phi_2\ E\ O) \tag{124}$$

6 Bases

The notion of a *basis* in the λ-calculus is analogous to the corresponding notion in linear algebra: A basis B for a vector space W is a finite set of vectors, whose closure under the operations of addition and scalar multiplication is equal to W. The situation in the λ-calculus is similar: A basis B for a set of terms W is a set of λ-terms (which need not be finite), whose closure, under the operation of application, is equal to W.

An implication of this similarity in the two notions of bases, is that the questions we will concern ourselves with, regarding bases in the λ-calculus, are similar to the questions that are of

[7] *Raymond Merrill Smullyan* [1919 –].

[8] In the actual derivation, a 'μ' operator is used to fix a variable in an expression, but expressions that contain 'μ' can be rewritten in terms of a fixed-point operator: $\mu x.M$ is equivalent to $(\Phi\ (\lambda x.M))$, for any fixed-point combinator Φ.

interest in linear algebra: *How do we represent an arbitrary λ-term as a combination of λ-terms in a given basis?* and *How small can a basis be if it is to span the set Λ^0 of all combinators?*

6.1 DEFINITION: *A Basis.* A set B is a *basis* for a set L if and only if for any $M \in \mathsf{L}$ there exists $N \in \mathsf{B}^+$, such that $M = N$.

6.2 DEFINITION: *Abstraction Algorithm.* Let a set B be a basis for a set L. An algorithm that associates with each $M \in \mathsf{L}$ a particular $N \in \mathsf{B}^+$ is known as an *abstraction algorithm.*

6.3 THEOREM: *The set* $\mathsf{N} = \{\mathbf{I}, \mathbf{K}, \mathbf{B}, \mathbf{C}, \mathbf{S}\}$ *forms a basis for* Λ^0.

Proof: Pick $M \in \Lambda^0$. We want to show that $M \in \mathsf{N}^+$. We use $[\![\ ]\!]_{\mathsf{N}}$ to denote the abstraction algorithm for N. The algorithm is defined inductively on the length of M. There are five cases to consider, following the recursive structure of a λ-term of the form $(\lambda\nu.(M\ N))$.

There are five cases to consider:

- Base case: $[\![x]\!]_{\mathsf{N}} = x$, for $x \in \{\mathbf{I}, \mathbf{K}, \mathbf{B}, \mathbf{C}, \mathbf{S}\} \cup \mathsf{Symbols}$.
- *The* $\mathbf{K}$*-Case:* $M = \lambda\nu.M'$, where $\nu \notin \mathsf{FreeVars}(M')$.
  We have $\|M'\| < \|M\|$, and
  $$((\lambda m\nu.m)\ M') \quad = \quad (\mathbf{K}\ M') \quad \longrightarrow\!\!\!\!\!\rightarrow \quad M$$
  So we define
  $$[\![\lambda\nu.M']\!]_{\mathsf{N}} \quad = \quad (\mathbf{K}\ [\![M']\!]_{\mathsf{N}}) \tag{125}$$
- *The* $\mathbf{C}$*-Case:* $M = \lambda\nu.(M'_\nu\ M'')$, where $\nu \in \mathsf{FreeVars}(M'_\nu), \nu \notin \mathsf{FreeVars}(M'')$.
  We have $\|\lambda\nu.M'_\nu\|, \|M''\| < \|M\|$, and
  $$\begin{array}{rl} & ((\lambda m'm''\nu.(m'\ \nu\ M''))\ (\lambda\nu.M'_\nu)\ M'') \\ = & (\mathbf{C}\ (\lambda\nu.M'_\nu)\ M'') \\ \longrightarrow\!\!\!\!\!\rightarrow & M \end{array}$$
  So we define
  $$[\![\lambda\nu.(M'_\nu\ M'')]\!]_{\mathsf{N}} \quad = \quad (\mathbf{C}\ [\![\lambda\nu.M'_\nu]\!]_{\mathsf{N}}\ [\![M'']\!]_{\mathsf{N}}) \tag{126}$$
- *The* $\mathbf{B}$*-Case:* $M = \lambda\nu.(M'\ M''_\nu)$, where $\nu \notin \mathsf{FreeVars}(M'), \nu \in \mathsf{FreeVars}(M''_\nu)$
  We have $\|M'\|, \|\lambda\nu.M''_\nu\| < \|M\|$, and
  $$\begin{array}{rl} & ((\lambda m'm''\nu.(m'\ (m''\ \nu)))\ M'\ (\lambda\nu.M''_\nu)) \\ = & (\mathbf{B}\ M'\ (\lambda\nu.M''_\nu)) \\ \longrightarrow\!\!\!\!\!\rightarrow & M \end{array}$$
  So we define
  $$[\![\lambda\nu.(M'\ M''_\nu)]\!]_{\mathsf{N}} \quad = \quad (\mathbf{B}\ [\![M']\!]_{\mathsf{N}}\ [\![\lambda\nu.M''_\nu]\!]_{\mathsf{N}}) \tag{127}$$
- *The* $\mathbf{S}$*-Case:* $M = \lambda\nu.(M_\nu\ N_\nu)$, where $\nu \in \mathsf{FreeVars}(M_\nu), \nu \in \mathsf{FreeVars}(N_\nu)$
  We have $\|\lambda\nu.M'_\nu\|, \|\lambda\nu.M''_\nu\| < \|M\|$, and
  $$\begin{array}{rl} & ((\lambda m'm''\nu.(m'\ \nu\ (m''\ \nu)))\ (\lambda\nu.M'_\nu)\ (\lambda\nu.M''_\nu)) \\ = & (\mathbf{S}\ (\lambda\nu.M'_\nu)\ (\lambda\nu.M''_\nu)) \\ \longrightarrow\!\!\!\!\!\rightarrow & M \end{array}$$
  So we define
  $$[\![\lambda\nu.(M_\nu\ N_\nu)]\!]_{\mathsf{N}} \quad = \quad (\mathbf{S}\ [\![\lambda\nu.M'_\nu]\!]_{\mathsf{N}}\ [\![\lambda\nu.M''_\nu]\!]_{\mathsf{N}}) \tag{128}$$

This completes the proof. ■

The **C**, and **B**-comb cases are, in fact, special instances of the **S**-case: The term $\lambda\nu.(M_\nu\ N)$ can be rewritten as $\lambda\nu.(M_\nu\ (\mathbf{K}\ N\ \nu))$, and the term $\lambda\nu.(M\ N_\nu)$ can be rewritten as $\lambda\nu.((\mathbf{K}\ M\ \nu)\ N_\nu)$. Furthermore, $\lambda\nu.\nu$ can be rewritten as $\lambda\nu.(\mathbf{K}\ \nu\ (\mathbf{K}\ \nu))$, so that the **S**-case will be applicable. This is the rationale behind Theorem 6.5.

6.4 EXAMPLES: Here are some familiar combinators, and their equivalents in N^+:

$$\begin{array}{rcccc} \mathbf{W} & = & \lambda xy.(x\ y\ y) & = & (\mathbf{C}\ \mathbf{S}\ \mathbf{I}) \\ \mathbf{Succ}_{\mathsf{Church}} & = & \lambda xyz.(y\ (x\ y\ z)) & = & (\mathbf{S}\ \mathbf{B}) \\ \ulcorner 0 \urcorner & = & \lambda xy.y & = & (\mathbf{K}\ \mathbf{I}) \end{array}$$

6.5 THEOREM: *The set* $\{\mathbf{K}, \mathbf{S}\}$ *forms a basis for* Λ^0.

Proof: Observe that:

$$\mathbf{I} = (\mathbf{S}\ \mathbf{K}\ \mathbf{K}) \tag{129}$$

$$\mathbf{B} = (\mathbf{S}\ (\mathbf{K}\ \mathbf{S})\ \mathbf{K}) \tag{130}$$

$$\mathbf{C} = (\mathbf{S}\ (\mathbf{S}\ (\mathbf{K}\ \mathbf{S})\ (\mathbf{S}\ (\mathbf{K}\ \mathbf{K})\ \mathbf{S}))\ (\mathbf{K}\ \mathbf{K})) \tag{131}$$

Now apply Theorem 6.3. ■

6.6 DEFINITION: *A One-Point Basis.* A one-point basis is a basis consisting of a single λ-term.

6.7 THEOREM: *There exists a one-point basis for* Λ^0.

Proof: Let X be the ordered pair of **S** and **K**, i.e.:

$$X = [\mathbf{S}, \mathbf{K}] = \lambda x.(x\ \mathbf{S}\ \mathbf{K}) \tag{132}$$

Observe that

$$(X\ (X\ (X\ X))) = \mathbf{K} \tag{133}$$

$$(X\ (X\ (X\ (X\ X)))) = \mathbf{S} \tag{134}$$

Now apply Theorem 6.5. ■

The λ-term X in (132), which forms the one-point basis used in the proof of Theorem 6.7, is the smallest (in the sense of *length* of the λ-term, see Definition 2.6) λ-term that forms a one-point basis, the author is aware of. This author has not found this λ-term in any of the standard literature on the λ-calculus, and so to the best of his knowledge, this λ-term is new.

7 Conclusion and Issues

In this work we provided a brief introduction to the λ-calculus. We covered the following topics:

- Syntax
- Reduction rules
- λ-Definability
- Fixed points
- Bases

This work provides a cursory overview of some of the main topics in the λ-calculus. For a more comprehensive treatment of the λ-calculus, we refer the reader to Church's *Calculi of Lambda Conversion* [4], Curry's *Combinatory Logic I, II* [5, 6], and Barendregt's *The Lambda Calculus: Its Syntax and Semantics* [1].

References

[1] Hendrik P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. North-Holland, 1984.

[2] Hendrik P. Barendregt. Functional programming and the λ-calculus. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 7, pages 323 – 363. MIT Press, Cambridge, Massachusetts, 1990.

[3] Hans Bekič. *Definable Operations in General Algebras, and the Theoryof Automata and Flowcharts*, pages 30–55. Lecture Notes in Computer Science, 177. Springer-Verlag, 1984.

[4] Alonzo Church. *The Calculi of Lambda-Conversion*. Princeton University Press, 1941.

[5] Haskell B. Curry, Robert Feys, and William Craig. *Combinatory Logic*, volume I. North-Holland Publishing Company, 1958.

[6] Haskell B. Curry, J. Roger Hindley, and Jonathan P. Seldin. *Combinatory Logic*, volume II. North-Holland Publishing Company, 1972.

[7] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

[8] Joseph Stoy. *Denotational Semantics: the Scott-Strachey approach to programming language theory*. The MIT Press Series in Computer Science. MIT Press, 1977.

[9] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, Massachusetts, 1993.