A Brief Introduction to the Lambda Calculus

Stuart A. Kurtz

Draft of March 29, 2004

1 Introduction

When you first learned about functions, they were most likely introduced as *abstractions* of *expressions.*

For example, consider the expression $2+3$, which can be immediately evaluated by the rules of elementary arithmetic to give value 5. We can *abstract*, i.e., generalize the notion of "adding 3 to 2" to that of "adding 3 to something," giving rise the function $f : x \mapsto x+3$. We can then *apply* this function to other arguments, e.g., $f(4) = 4+3 = 7$. This style of function definition is *intentional*—functions are defined by specifying an intended means of evaluation.

Later, you may have been encouraged to think of functions *extensionally*, i.e., as sets of ordered pairs, e.g. $f = \{(x,y) : \ y = x+3\}$. Such an approach conveys substantial metamathematical advantages, but it comes at a terrible cost—functions are no longer things that you can compute with.

The lambda calculus [Chu41] returns to the notion of functions as abstractions of expressions. Abstraction is accomplished by the eponymous lambda (λ), by means of which we could define the function f as $\lambda x.\ x+3$. An abstraction may be *applied* to an argument, e.g.,

$$\begin{aligned} f\ 2 &\rightarrow (\lambda x.\ x+3)\ 2 \\ &\rightarrow 2+3 \\ &\rightarrow 5 \end{aligned}$$

Note that parenthesis are used to indicated grouping, the scope of the λ is considered to extend as far as possible while remaining consistent with the parentheses, and that application is represented by juxtaposing the function and its argument. Also note that the use of parentheses is extremely sparing—with experience, fewer parentheses rather than more improve readability.

For students of Lisp, this familiar example would be expressed using a syntax in where the use of parenthesis is strictly regulated:

$$\begin{aligned} ((\texttt{lambda}\ (\texttt{x})\ (+\ \texttt{x}\ \texttt{2}))\ \texttt{3}) &\rightarrow (+\ \texttt{2}\ \texttt{3}) \\ &\rightarrow \texttt{5}. \end{aligned}$$

One of the nicest features of the lambda notation for functional abstraction is that it can be used equally well to describe *higher order* functions, i.e., functions that take functions as arguments, and return functions as results. For example, consider the composition function $f \circ g: \; x \mapsto f(g(x))$. Properly understood, composition ($\circ$) is a binary function that takes functional two arguments (f and g), and produces another function. Thus, we could define $\circ = \lambda f, g.\ \lambda z.\ f\ (g\ z)$, so that

$$\begin{aligned} \circ\, (f, g)\ x &\rightarrow (\lambda f, g.\ \lambda z.\ f\ (g\ z))\ (f, g)\ x \\ &\rightarrow (\lambda z.\ f\ (g\ z))\ x \\ &\rightarrow f\ (g\ x). \end{aligned}$$

There are a few things to note about this. Application associates to the left, thus, for example we can leave out parentheses in $\circ\ (f, g)\ x$, which is interpreted as $(\circ\ (f, g))\ x$, but they are mandatory in $f\ (g\ x)$. Multi-ary functions are handled by having lambda bind a comma-separated list of arguments, and we introduce tupling in order create multi-ary arguments. The body of a lambda expression can itself be a lambda expression, which is how a function can return a function.

We could do it that way, but the designers of the lambda calculus approached matters somewhat differently. One of the guiding principles in designing a mathematical theory is *parsimony*, that is the principle that the undefined (basic) notions of the theory should be as simple, and as few in number, as possible. Handling multi-ary functions as above requires a number of complications: lambda has to be able to bind multiple variables, and we have to introduce the notion of tupling to the language. A more parsimonious approach is via *currying*, in which a binary function is viewed as a unary function that consumes the first argument, and which returns a functional result that consumes the second argument. Thus, the actual definition of composition in the lambda calculus is $\circ = \lambda f.\ \lambda g.\ \lambda x.\ f\ (g\ x)$, and we have

$$\begin{aligned} \circ\ f\ g\ x &\rightarrow (\lambda f.\ \lambda g.\ \lambda z.\ f\ (g\ z))\ f\ g\ x \\ &\rightarrow (\lambda g.\ (\lambda z.\ f\ (g\ z))\ g\ x \\ &\rightarrow (\lambda z.\ f\ (g\ z))\ x \\ &\rightarrow f\ (g\ x). \end{aligned}$$

A second application of parsimony comes in answering the following question: "if lambda-terms denote functions, what is the domain and range of these functions?" The answer is already hinted at in the composition example: the domain and range of lambda-defined functions are lambda-terms.

A third application of parsimony is in the choice of expressions over which we can abstract. The solution to the domain and range question anticipates the solution to the expression question: in the *pure* lambda calculus, we generalize over lambda-terms.

2 A Formal Development

Definition 1 (λ-terms) *We begin by positing the existence of an infinite set of atomic variables, x, y, z, ... We then define the set Λ of λ-terms as follows:*

- *every variable is a λ-term,*
- *if M and N are λ-terms, then $apply(M, N)$ is a λ-term, and*
- *if x is a variable, and M is a λ-term, then $lambda(x, M)$ is a λ-term.*

2.1 What is a λ-term?

Definition 1 looks quite different from the informal presentation in Section 1. Indeed, it differs from traditional approaches [Bar84, HS86]. The following discussion should make the relationship clearer.

A term in a term-algebra should be thought of a tree. For example, the term that represents composition should not be thought of as the string of characters "$\lambda f.\ \lambda g.\ \lambda x.\ f\ (g\ x)$," but rather graphically as in Figure 1. The problem with such graphical forms is their unwieldiness, both typographically and computationally. But this point should not be forgotten: even though we will often speak of strings such as "$\lambda f.\ \lambda g.\ \lambda x.\ f\ (g\ x)$" as if they were λ-terms, they are not λ-terms. They are convenient, terse ways to represent λ-terms[1].

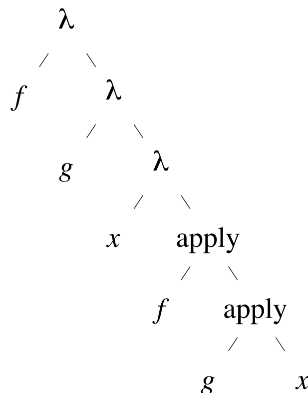

Figure 1: Graphical representation of the λ-term for composition.

[1]The fastidious will point out that graphical objects such as Figure 1 are not λ-terms either. They're simply another representation: λ-terms are irreducibly abstract objects of mathematical discourse.

The point must be granted, but not dwelled upon. The correspondence between graphical representations of abstract terms, and the abstract terms themselves, is simple and direct, whereas the correspondence between textual representations and λ-terms relies on various essentially arbitrary conventions, such as the scope rules and the left-associativity of application.

If in the end it seems that we're begging the question of what λ-terms are, it is because we are: abstract notions are abstract, and sustaining the attempt to reify them is foolish.

2.2 The Interpretation of Inductive Definitions

At this point, we return to Definition 1, which is written in the standard mathematical style, and whose interpretation requires understanding the conventions of that style. A complete discussion of these issues requires an excursion into descriptive set theory [Mos79], so our treatment is necessarily sketchy.

A possible objection to Definition 1 is that it is *impredicative*: the notion to be defined (λ-terms) is defined in terms of itself. Impredicative definitions can be dangerous, e.g., Russell's paradox (a paradox of naïve set theory) rests on an impredicative definition. But as regards Definition 1, there is considerable mathematical experience that defining a set in terms of a countable initial set (the variables), and a countable set of finitary functions (abstraction, application) is safe. Indeed, such definitions are called *inductive*, and it can be proven that if arithmetic—the simplest non-trivial inductive theory—is consistent, then all inductive definitions are free from paradox.

Another objection is that Definition 1 is not a *well-definition*, i.e., there may be multiple sets that satisfy the definition, or perhaps none. The objection is valid: there are in fact infinitely many sets that satisfy the definition. So what does the definition mean? The mathematical convention is that the clauses in such a definition are *exclusive* and *exhaustive*, and so every element of Λ is a variable, an abstraction, or an application, in precisely one way. Again, assuming that arithmetic is consistent, it can be shown that inductive definitions are well-definitions.

2.3 β-Reduction

At this point, we've only given the abstract syntax of λ-terms. To make it into an interesting mathematical theory, we have to show how that syntax can be manipulated. The point of rule-based definition of function is that you can evaluate the function at an argument by applying its rule. If we think of abstractions (terms of the form $\lambda x.\ M$) as rule-base definitions of functions, then applications of abstractions (terms of the form $(\lambda x.\ M)\ N$) cry out for evaluation. In the λ-calculus, this process of evaluation is called β-reduction.

Definition 2 (Redex) *A* redex *is a λ-term that consists of the application of an abstraction.*

Redexes are simply the places in a λ-term where the application of a rule-based function to an argument can be expanded in terms of the rule. The expansion of the rule takes the form of substituting a λ-term N for a variable x in another λ-term M, written $M[x := N]$.

Definition 3 (β-reduction) *Let P and Q be λ-terms such that Q arises from P by replacing a subterm $(\lambda x.M)N$ of P with $M[x := N]$. Then we say that P β-reduces in one step to Q, and write $P \to_1 Q$. If Q arises from P by a sequence of one-step β-reductions, then we say that P β-reduces to Q, and write $P \to Q$.*

Consider the lambda term $(\lambda x.\ \lambda y.\ (\lambda z.\ z\ z)\ x\ y)\ a$. This term contains two redexes. The entire term is a redex, consisting of the application of $\lambda x.\ \lambda y.\ (\lambda z.\ z\ z)\ x\ y$ to a. There is also an *inner redex*: $(\lambda z. z\ z)\ x$. Thus, evaluation can take two different paths:

$$\begin{array}{rcl} (\lambda x.\ \lambda y.\ (\lambda z.\ z\ z)\ x\ y)\ a & \rightarrow & \lambda y.\ (\lambda z.\ z\ z)\ a\ y \\ & \rightarrow & \lambda y.\ a\ a\ y, \end{array}$$

and

$$\begin{array}{rcl} (\lambda x.\ \lambda y.\ (\lambda z.\ z\ z)\ x\ y)\ a & \rightarrow & (\lambda x. \lambda y.\ x\ x\ y)\ a \\ & \rightarrow & \lambda y.\ a\ a\ y. \end{array}$$

There are a couple of important things to observe about these reduction sequences. First, both reduction sequences end with *normal forms*, i.e., λ-terms that do not contain a redux, and therefore can be evaluated any further. Normal forms are completed computations, and just as in real-world programming, there are λ-terms that cannot be reduced to normal form. These terms can be thought of as specifying non-terminating computations. Second, although the reduction sequences are quite different, both terminated in the same normal form. This is not an accident.

Theorem 4 (Confluence) *Let $M \rightarrow N$ mean that there is a sequence of β-reductions that convert M to N. Suppose $M \rightarrow N_0$ and $M \rightarrow N_1$. Then there is a λ-term Q such that $N_0 \rightarrow Q$ and $N_1 \rightarrow Q$.*

Intuitively, the confluence theorem says that if two computations beginning with the same term diverge, they can be brought back together by β-reduction. The proof of the confluence theorem is well beyond the scope of this introduction, but it can be found in any of the standard references [Bar84, HS86], sometimes under the name of the "diamond property." The confluence theorem has several substantial corollaries.

Corollary 5 (Uniqueness of Normal Forms) *Suppose $M \rightarrow N_0$ and $M \rightarrow N_1$, where N_0 and N_1 are both normal forms. Then $N_0 = N_1$.*

The corollary follows directly from the theorem, as if N is a normal form, and $N \rightarrow P$, then $N = P$.

Definition 6 (Equivalence) Equivalence *(in notation $\equiv$) is the transitive, reflexive, symmetric closure of β-reduction.*

Corollary 7 (The Church-Rosser Theorem) *Let M and N be λ-terms. Then $M \equiv N$ if and only if there is a λ-term P such that $M \rightarrow P$ and $N \rightarrow P$.*

The λ-calculus is essentially the theory of λ-terms under equivalence.

2.4 Climbing Out of the Trap

Our discussion of β-reduction has been naïve, or less pejoratively, incomplete, in two different senses.

The first sense of incompleteness comes from the fact that the choice of name for an abstracted variable is essentially arbitrary. For example, $I_x = \lambda x.x$ is the identity function, but so too is $I_z = \lambda z.\ z$. In this case, the particular choice of name x or z for the abstracted variable is arbitrary, and there is no reason to distinguish I_x from I_z. The consistent change of name of an abstracted variable is called α-reduction, and we view α-reductions as "free." Thus, when we write $N_0 = N_1$ in Corollary 5, equality means "up to α-convertibility."

The more substantial incompleteness is that we've glossed over what it means to substitute a λ-term N for a variable x in a λ-term M. Superficially, substitution seems pretty straightforward.

Definition 8 (Inconsistent Substitution) *Let M and N be λ-terms, and let x be a variable. We define the result of substituting N for x in M, written $M[x := N]$, by induction on the structure of λ-terms as follows:*

1. $x[x := N] = N$,
2. $y[x := N] = y$,
3. $(\lambda x.P)[x := N] = (\lambda x.P)$,
4. $(\lambda y.P)[x := N] = \lambda y.(P[x := N])$,
5. $(P\ Q)[x := N] = (P[x := N])\ (Q[x := N])$,

where in clauses 2 and 4, y plays the role of any variable distinct from x.

Before we get on with criticizing Definition 8, we should stop and consider what is new and right about the form of the definition. In effect, Definition 8 defines a function (by abuse of notation, $\lambda z.\ z[x := N]$), whose domain is an inductively defined set (the set of λ-terms). The definition of the function follows the definition of the inductive set, with clauses 1 and 2 defining substitution into variables, clauses 3 and 4 defining substitution into applications, and clause 5 defining substitution into compositions. This style of function definition is safe, and it can be justified in essentially the same way as an inductive definition. Indeed, functional programmers are very familiar with this style of *natural recursion* [FFFK02].

The problem with Definition 8 is with clause 4. Consider $(\lambda y.\ x)\ z\ N$, where N is an arbitrary λ-term.

$$\begin{aligned}(\lambda x.\ \lambda y.\ x)\ z\ N &\rightarrow (\lambda y.\ z)\ N \\ &\rightarrow z.\end{aligned}$$

This seems entirely straightforward. But we've already argued that systematic changes of abstracted variables shouldn't matter, thus $\lambda x.\ \lambda y.\ x = \lambda x.\ \lambda z.\ x$ (note that we've changed y for z). Now,

$$\begin{aligned} (\lambda x.\ \lambda z.\ x)\ z\ N &\rightarrow (\lambda z.\ z)\ N \\ &\rightarrow N. \end{aligned}$$

If this stands, it means that $N \equiv z$, without reference to the structure of N. Thus, if M is another λ-term, essentially the same example shows $M \equiv z$, and hence $M \equiv N$, and so there is only one equivalence class of terms, which is neither interesting nor useful.

The problem comes in the step

$$(\lambda x.\ \lambda z.\ x)\ z \rightarrow \lambda z.\ z.$$

What happens is that the unbound, second occurrence of z in the left-hand side, becomes bound as a result of the substitution. In effect, the meaning of the "passive" argument z changed through the substitution process. This is called *the inadvertent capture of a free variable*, and it is not our intent.

The solution to this quandary is to limit clause 4 to the case where y does not have any unbound occurrences in N:

Definition 9 (Substitution) *Let M and N be λ-terms, and let x be a variable. We define the result of substituting N for x in M, written $M[x := N]$, by induction on the structure of λ-terms as follows:*

1. $x[x := N] = N$,
2. $y[x := N] = y$,
3. $(\lambda x.P)[x := N] = (\lambda x.P)$,
4. $(\lambda y.P)[x := N] = \lambda y.(P[x := N])$,
5. $(P\ Q)[x := N] = (P[x := N])\ (Q[x := N])$,

where in clauses 2 and 4, y plays the role of any variable distinct from x, and where in clause 4, we require that y does not have any unbound occurrences in N.

Is this satisfactory? Initially, the answer seems to be "no," because our definition is no longer complete: we seem not to define $(\lambda y.P)[x := N]$ in the case where y has free occurrences in N. But we're forgetting about our convention that the choice of binding variable in an abstraction is essentially arbitrary. If y has free occurrences in N, we're permitted to replace it in $\lambda y.\ P$ by a variable y' that doesn't, thus

$$\begin{aligned} (\lambda y.\ P)[x := N] &= (\lambda y'.\ P[y := y'])[x := N] \\ &= \lambda y'.\ (P[y := y'][x := N]) \end{aligned}$$

Getting the definition of substitution right, in this context or any other, is tricky, and it has often been done incorrectly. One is tempted to say that the examples that show problems with the incorrect definitions are contrived, but experience shows otherwise[2].

3 Computing in the λ-Calculus

Lisp is a general purpose programming language, and the connections between Lisp and the λ-calculus should be obvious to those familiar with both. This argues that it should be possible to use the λ-calculus as a means of specifying and carrying out computations. Of course, this argument gets historical causality wrong: it was the usefulness of the λ-calculus as a means of specifying and carrying out computations that lead McCarthy [McC60] to implement them as a concrete programming language.

But real-world programming languages, like Lisp, have facilities that are not obviously present in the pure λ-calculus: they have primitive datatypes for things like the integers, they have ways to create composite types (tuples, records, vectors, etc.), and they have control structures such as conditionals and recursion. It is not immediately whether or not analogs to these facilities can be found in the λ-calculus.

3.1 Tupling and Untupling

A place to begin is with simple pairs. We can represent the pair (M_0, M_1) by $\lambda z.\ z\ M_0\ M_1$. From this, we can define a pairing function $\tau = \lambda x.\ \lambda y.\ \lambda z.\ z\ x\ y$. The projection functions are $\pi_0 = \lambda p.\ p\ (\lambda x.\ \lambda y.\ x)$ and $\pi_1 = \lambda p.\ p\ (\lambda x.\ \lambda y.\ y)$.

Exercise 1 *Draw a graphical representation, as in Figure 1 for the pairing function τ.*

---

[2]Most famously, when John McCarthy first defined the Lisp programming language based on the λ-calculus, he described the evaluation process through equations that essentially embodied the incorrect definition of substitution.

By the time it was realized that a mistake had been made, Lisp programmers had come to rely on the behavior—many of the captures of a free variable were intentional. This naturally lead to a reluctance to view what Lisp did as incorrect, and the ostensibly non-normative terminology of *dynamic scope* was used to describe what Lisp did, whereas *static scope* was used to describe what the λ-calculus (and programming languages in the Algol family) did. The primary advantage of dynamic scope was that it enabled a programming idiom in which the value of a global variable could be locally overridden. This meant that a procedure could in effect take a very large number of implicit arguments which were given default values in the global environment, and overridden as needed. The primary disadvantage of dynamic scope was that it was no longer sufficient to understand what a procedure was intended to do to understand how using it would impact your program: you also had to understand how it was implemented.

The argument over dynamic vs. static scope was sustained until compliation became an important means for speeding up Lisp programs. During compliation, the abstracted variables become associated with storage locations rather than names, and capture (intentional or not) does not occur. As a result, Lisp programmers discovered that while interpreted programs used dynamic scope, the compiled versions of the same programs used static scope. As compliation technologies improved, it became faster to execute a single line of code by compiling it and executing the compiled code than by simple interpretation.

Soon thereafter, Lispers threw in the towel, and modern versions of Lisp [KCE98, Ste84] use static scope, i.e., correct substitution rules.

We verify that the pairing function works as intended:

$$\begin{array}{rcl} \tau\ M_0\ M_1 & = & (\lambda x.\ \lambda y.\ \lambda z.\ z\ x\ y)\ M_0\ M_1 \\ & \rightarrow & (\lambda y.\ \lambda z.\ z\ M_0\ y)\ M_1 \\ & \rightarrow & \lambda z.\ z\ M_0\ M_1 \\ & = & (M_0, M_1), \end{array}$$

and that π_0 works as intended:

$$\begin{array}{rcl} \pi_0\ (M_0, M_1) & = & (\lambda p.\ p\ (\lambda x.\ \lambda y.\ x))\ (M_0, M_1) \\ & \rightarrow & (M_0, M_1)\ (\lambda x.\ \lambda y.\ x) \\ & = & (\lambda z.\ z\ M_0\ M_1)\ (\lambda x.\ \lambda y.\ x) \\ & \rightarrow & (\lambda x.\ \lambda y.\ x)\ M_0\ M_1 \\ & \rightarrow & (\lambda y.\ M_0)\ M_1 \\ & \rightarrow & M_0 \end{array}$$

Exercise 2 *Show that* $\pi_1\ (M_0, M_1) \rightarrow M_1$.

Arbitrary n-tuples can be dealt with in an analogous fashion. For example, if $n = 4$, we can represent (M_0, M_1, M_2, M_3) by $\lambda z.\ z\ M_0\ M_1\ M_2\ M_3$. We then have a 4-tupling function τ_4, and projection functions $\pi_{4,0}$, $\pi_{4,1}$, $\pi_{4,2}$ and $\pi_{4,3}$, by analogy with pairing.

3.2 Conditionals

We can represent **true** by $\lambda x.\ \lambda y.\ x$, and **false** by $\lambda x.\ \lambda y.\ y$. We can represent the construction `if` M `then` P `else` Q by $M\ P\ Q$.

Exercise 3 *Show that the* `if` ... `then` ... `else` ... *construction has the intended reduction properties, i.e.,*

$$\texttt{if}\ \textbf{true}\ \texttt{then}\ P\ \texttt{else}\ Q \quad \rightarrow \quad P,$$

and

$$\texttt{if}\ \textbf{false}\ \texttt{then}\ P\ \texttt{else}\ Q \quad \rightarrow \quad Q,$$

3.3 Numbers

We define $f^n\ x$, the n-fold iteration of f at x, by induction on n as follows:

- $f^0\ x = x$,
- $f^{n+1}\ x = f\ (f^n\ x)$.

A definition like this gives us a simple recipe for constructing $f^n x$ for any n, e.g.,

$$\begin{aligned} f^3\ x &= f\ (f^2\ x) \\ &= f\ (f\ (f^1\ x)) \\ &= f\ (f\ (f(f^0\ x))) \\ &= f\ (f\ (f\ x)) \end{aligned}$$

Definition 10 (Church numerals) *Let n be a natural number, i.e., an element of the set $\{0, 1, 2, \ldots\}$ of non-negative whole numbers. The Church numeral $\lceil n \rceil = \lambda f.\ \lambda x.\ f^n\ x$.*

Church numerals are λ-terms, which can be used to represent the natural numbers in the λ-calculus. For this to be useful, we need to show that there are functions that implement the standard number theoretic functions on the natural numbers. For example, consider the successor function: $S\ n = n + 1$. Our obligation is to construct a λ-term $\lceil S \rceil$ such that $\lceil S \rceil\ \lceil n \rceil \rightarrow \lceil n + 1 \rceil$. We chose $\lceil S \rceil = \lambda n.\ \lambda f.\ \lambda x. n\ f\ (f\ x)$. Now:

$$\begin{aligned} \lceil S \rceil \lceil n \rceil &= (\lambda n.\ \lambda f.\ \lambda x. n\ f\ (f\ x))\ \lceil n \rceil \\ &\rightarrow \lambda f.\ \lambda x.\ \lceil n \rceil\ f\ (f\ x) \\ &\rightarrow \lambda f.\ \lambda x.\ f^n\ (f\ x) \\ &\rightarrow \lambda f.\ \lambda x.\ f^{n+1} x \\ &= \lceil n + 1 \rceil, \end{aligned}$$

as required.

For addition, we need a λ-term $\lceil + \rceil$ such that $\lceil + \rceil \lceil m \rceil \lceil n \rceil \rightarrow \lceil m + n \rceil$. One such term is $\lceil + \rceil = \lambda m.\ \lambda n.\ \lambda f.\ \lambda x.\ m\ f\ (n\ f\ x)$. We leave the verification to as an exercise.

Exercise 4 *Show that $\lceil + \rceil\ \lceil m \rceil\ \lceil n \rceil \rightarrow \lceil m + n \rceil$.*

Exercise 5 *Construct a λ-term $\lceil \times \rceil$ such that $\lceil \times \rceil\ \lceil m \rceil\ \lceil n \rceil \rightarrow \lceil m \times n \rceil$.*

Exercise 6 *Show that there is a λ-term $\lceil zero? \rceil$ that computes the "equal to zero" predicate.*

$$\lceil \mathit{zero?} \rceil\ \lceil n \rceil \rightarrow \begin{cases} \textbf{true}, & \text{if } n = 0 \\ \textbf{false}, & \text{if } n > 0. \end{cases}$$

There is an interesting story regarding Church numerals. Church numbers were defined, and their properties first explored, by the mathematician Alonzo Church and his Ph.D. student Stephen Kleene. The λ-terms $\lceil S \rceil$, $\lceil + \rceil$, and $\lceil \times \rceil$ were quickly discovered, but subtraction over proved much more difficult. They tried for a long time to find a λ-represention for subtraction, without success. Finally, while sitting in the dentist's chair, Kleene had a flash of inspiration and realized how it could be done. His approach lead in time to a much deeper understanding of the λ-calculus, and ultimately to another formal system for computing over the natural numbers called the Kleene μ-calculus, which is a progenitor of algebraic programming languages like C, Algol, Pascal, and Java.

We begin by considering the predecessor function.

Definition 11 (Predecessor) *The predecessor function P is defined as follows:*

- $P\ 0 = 0,$
- $P\ (S\ n) = n.$

Consider the function $\iota : (x, y) \mapsto (y, S\ y)$.

Exercise 7 *Show that ι is λ-definable, i.e., there exists a λ-term $\lceil \iota \rceil$ such that $\lceil \iota \rceil\ (x, y) \to (y, \lceil S \rceil\ y)$.*

We can prove by induction on n that $\iota^n\ (0, 0) = (P\ n, n)$, and therefore that $\pi_0\ \iota^n(0, 0) = P\ n$. We now define $\lceil P \rceil = \lambda n.\ \pi_0\ (n\ \lceil \iota \rceil\ (\lceil 0 \rceil, \lceil 0 \rceil))$, and see

$$\begin{aligned}
\lceil P \rceil \lceil n \rceil &= (\lambda n.\ \pi_0\ (n\ \lceil \iota \rceil\ (\lceil 0 \rceil, \lceil 0 \rceil))\ \lceil n \rceil \\
&\to \pi_0\ (\lceil n \rceil\ \lceil \iota \rceil\ (\lceil 0 \rceil, \lceil 0 \rceil)) \\
&\to \pi_0\ (\lceil \iota \rceil^n\ (\lceil 0 \rceil, \lceil 0 \rceil)) \\
&\to \pi_0\ (\lceil P\ n \rceil, \lceil n \rceil) \\
&\to \lceil P\ n \rceil
\end{aligned}$$

The approach is a familiar use of *register-based programming* for students familiar with Scheme, especially as covered in Felliesen, et. al. [FFFK02]. Indeed, with the right hints, and having seen a few well chosen examples, this is a program that a reasonably talented Scheme student would be expected to discover. But this is no knock on Kleene—he had no hints, and no examples. There had to be a first register-based program, and this was it.

Definition 12 (Monus) *Let m and n be natural numbers. We define the* monus ($\dot{-}$) *function as*

$$m \dot{-} n = \begin{cases} m - n, & \text{if } m \geq n \\ 0, & \text{if } m < n. \end{cases}$$

Monus is a version of subtraction with a domain truncated to the natural numbers.

It is easy to see that $m \dot{-} n = P^n\ m$. From this, the λ-definability of $\dot{-}$ follows almost immediately.

Exercise 8 *Show that there is a λ-term $\lceil \dot{-} \rceil$ such that*

$$\lceil \dot{-} \rceil\ \lceil m \rceil\ \lceil n \rceil \to \lceil m \dot{-} n \rceil.$$

The subsequent development of λ-computability over the Church numerals was extremely rapid, and all of the standard number-theoretic functions—exponentiation, Euler's ϕ function, etc.—were soon shown to be λ-computable. Indeed, these are contained in the much larger, and mathematically important class of primitive recursive functions, and it was shown that all of the primitive recursive functions are λ-computable. When it was shown that Ackermann's function (a computable function that is not primitive recursive) is also λ-computable, Church made the following famous conjecture:

Conjecture 13 (Church's Thesis) *If f is a computable number-theoretic function, then there is a λ-term $\lceil f \rceil$ such that for all $n \in dom(f)$,*

$$\lceil f \rceil \; \lceil n \rceil \rightarrow \lceil f\ n \rceil.$$

This is a powerful conjecture, as it amounts to a claim that the λ-calculus is universal, at least with respect to computation over the natural numbers. The impact comes from the fact that for Church, the term "computable" is not defined, instead it is to be interpreted in its intuitive sense—there is a finite set of rules, which, if followed by a limited deterministic agent for finitely many steps on a given input n, results in the computation of $f\ n$.

References

[Bar84] Hendrik Peter Barendregt. *The Lambda Calculus*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. Elsevier Science Publishers B. V., Amsterdam, 1984.

[Chu41] Alonzo Church. *The Calculi of Lambda Conversion*, volume 6 of *Annals of Mathematics Studies*. Princeton University Press, Princeton, New Jersey, 1941.

[FFFK02] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthy. *How to Design Programs*. The MIT Press, Cambridge, Massachusetts, 2002.

[Göd31] Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I. *Monatshefte für Mathematik und Physik*, 38, 1931. Translated in von Heijenoort [vH67].

[HS86] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and the λ-calculus*, volume 1 of *London Mathematical Society Student Texts*. Cambridge University Press, Cambridge, United Kingdom, 1986.

[KCE98] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[McC60] John L. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3(4):184–195, 1960.

[Mos79] Yiannis N. Moschovakis. *Descriptive Set Theory*. Elsevier Science Ltd, Amsterdam, 1979.

[Ste84] Guy Steele. *Common Lisp: The Language*. Digital Press, Woburn, Massachusetts, second edition, 1984.

[vH67] Jean von Heijenoort. *From Frege to Gödel: A Source Book in Mathematical Logic*. Harvard University Press, Cambridge, Massachusetts, 1967.