JFP, 13 pages, 2020.
doi:10.1017/S095679682000009X

FUNCTIONAL PEARLS

Many more predecessors: A representation workout

OLEG KISELYOV
Tohoku University, Japan
(*e-mail:* oleg@okmij.org)

Abstract

From the outset, lambda calculus represented natural numbers through iterated application. The successor hence adds one more application, and the predecessor removes. In effect, the predecessor un-applies a term—which seemed impossible, even to Church. It took Kleene a rather oblique glance to sight a related representation of numbers, with an easier predecessor. Let us see what we can do if we look at this old problem with today's eyes. We discern the systematic ways to *derive* more predecessors—smaller, faster, and sharper—while keeping all teeth.

1 Introduction

Lambda calculus is banal in its operation—and yet is an unending source of delightful puzzles. One of the first was the predecessor: applied to the term representing a natural number $n + 1$, it should reduce to the representation of n. When the number n is represented as an n-times repeated application, the predecessor amounts to an un-application—which is not the operation lambda calculus supports. As Church was about to give up the hope of expressing arithmetic, his student Kleene was getting his wisdom teeth extracted, and under anesthetic (or so Barendregt, 1997 says) foreglimpsed the solution.

The tooth-wrenching story and Kleene's predecessor have become a part of the Functional Canon, told and retold in tutorials and textbooks, and invariably called "very tricky". I can sympathize, having searched for, and eventually finding, a different predecessor back in 1992. Incidentally, I also had a tooth extracted that year.

This article shows that by looking at the puzzle as a representation-change problem we see, in plain sight, more and more solutions—insightful, easier to explain and to write down on a single line, and to extend beyond numbers. We even spot an un-application.

2 Preliminaries

In this paper, we use the *pure* lambda calculus, whose expressions (also called terms) are made only of variables, abstractions, and applications, as defined below. (The

meta-variable e stands for an arbitrary expression and meta-variables x, y, z stand for arbitrary variables; $e_1[x := e_2]$ is the capture-avoiding substitution of x with e_2 in e_1):

Variables	$x, y, z ::=$	single letters, possibly with sub- and superscripts, excluding e but including x, y, z
Expressions	$e ::=$	$x \mid \lambda x.e \mid e\ e$
Reductions	$\rightsquigarrow_\beta$	$(\lambda x.e_1)\ e_2 \rightsquigarrow_\beta e_1[x := e_2]$

We take application to be left-associative, which lets us write repeated applications, such as $(e_1\ e_2)\ e_3$ without parentheses. Expressions like $\lambda x.e_1\ e_2$ are to be understood as $\lambda x.(e_1\ e_2)$: the body of an abstraction extends as far to the right as possible; parentheses delimit it if needed. Sometimes we write repeated abstractions like $\lambda x.\lambda y.e$ as $\lambda xy.e$.

We do not use types. Incidentally, predecessor, in any form, cannot be represented in simply typed lambda calculus, in principle: Statman (1979).

We write $e_1 \rightsquigarrow e_2$ for the compatible closure of $\rightsquigarrow_\beta$: the smallest relation containing $\rightsquigarrow_\beta$ with the property that if $e_1 \rightsquigarrow e_2$ then likewise $(\lambda x.e_1) \rightsquigarrow (\lambda x.e_2)$, $e\ e_1 \rightsquigarrow e\ e_2$ and $e_1\ e \rightsquigarrow e_2\ e$. We write $\rightsquigarrow^*$ for the transitive reflexive closure of $\rightsquigarrow$ and say that e_1 *reduces* to e_2 just in case the relation $e_1 \rightsquigarrow^* e_2$ holds. The reflexive, transitive, compatible, and symmetric closure of $\rightsquigarrow_\beta$ (i.e., $\rightsquigarrow_\beta$ congruence) is written $\doteq$; the expressions so related are called *equal*.

The pure lambda calculus has no constants or operations. To make its expressions easier to read and write, we shall refer to some terms by short and meaningful names. The name assignment (i.e., definitions) and the names themselves are not part of the calculus but a mere syntax sugar.[1] Here are sample definitions, for the expressions representing Booleans, ordered pairs, and composition:

$$\mathsf{id} := \lambda x.x \qquad \mathsf{pair} := \lambda x.\lambda y.\lambda p.p\ x\ y$$

$$\mathsf{true} := \lambda x.\lambda y.x \qquad \mathsf{fst} := \lambda p.p\ \mathsf{true}$$

$$\mathsf{false} := \lambda x.\lambda y.y \qquad \mathsf{snd} := \lambda p.p\ \mathsf{false}$$

$$\mathsf{and} := \lambda p.\lambda q.p\ q\ \mathsf{false} \qquad \mathsf{comp} := \lambda fg.\lambda x.f(g\ x)$$

As further notational convenience, we write $\lambda x.f(g\ x)$ as $f \circ g$ and $\lambda p.p\ e_1\ e_2$ as (e_1, e_2).[2] It is easy to see that $\mathsf{fst}\ (e_1, e_2) \doteq e_1$ and $\mathsf{snd}\ (e_1, e_2) \doteq e_2$ for arbitrary e_1 and e_2—as expected of pairs. We define the size of a term as the total count of its variables, applications, and lambdas. For example, the size of pair is 8.

The symbol $:=$ gives the name to the term on its right-hand side *as written* (modulo the renaming of bound variables, invoked implicitly as needed). It is common to name only normal forms of terms, noting exceptions explicitly. However, we often want to convey how the defined term is put together, from applications of other terms. In such cases, we

[1] "... the definitions are not part of our subject, but are, strictly speaking, mere typographical conveniences.... In spite of the fact that definitions are theoretically superfluous, it is nevertheless true that they often convey more important information than is contained in the propositions in which they are used. ... The collection of definitions embodies our choice of subjects and our judgement as to what is most important. Secondly, ... the definition contains an analysis of a common idea, and may therefore express a notable advance." (Whitehead & Russell, 1910) (p. 12)

[2] (e_1, e_2) is clearly equal to $\mathsf{pair}\ e_1\ e_2$. Also, (e_1, e_2) is in normal form whenever e_1 and e_2 are.

use the notation $\mathtt{name} :\doteq e$, to be read as giving name to the normal form of e, thereby asserted to exist.

Natural numbers are commonly represented in lambda calculus by means of an iterated application as shown below. We notate these so-called Church numerals as c_n, for the numeral representing the number n:

$$\mathsf{c}_0 := \lambda f.\lambda x.x \qquad \mathsf{c}_1 := \lambda f.\lambda x.f\,x \qquad \mathsf{c}_2 := \lambda f.\lambda x.f\,(f\,x) \qquad \ldots$$

We will also write c_n as $\lambda f.\lambda x.f^{(n)}x$, taking $f^{(n)}x$ to mean the n-times repeated application of f to x. A simple inductive demonstration, or just writing it out, shows that:

$$\lambda fx.f^{(n+1)}x \quad \doteq \quad \lambda fx.\mathsf{c}_{n+1}\,f\,x \quad \doteq \quad \lambda fx.\mathsf{c}_n f\,(f\,x) \quad \doteq \quad \lambda fx.f\,(\mathsf{c}_n f\,x) \qquad (1)$$

which leads us to the successor—a term whose application to c_n reduces to c_{n+1}. Equation (1) gives two such terms (we will be using the second one: the choice is arbitrary.):

$$\mathsf{succ}' := \lambda n.\lambda fx.n\,f\,(f\,x) \qquad \mathsf{succ} := \lambda n.\lambda fx.f\,(n\,f\,x)$$

The problem is to find the predecessor—a term pred such that the application $\mathsf{pred}\,\mathsf{c}_{n+1}$ reduces to c_n. (What should be the result of $\mathsf{pred}\,\mathsf{c}_0$ is an open choice; often it is c_0.)

We shall derive many predecessors, some known, most new, by contemplating the koan (*) below and following three trails of thought as they unfold. Finally, in Section 8, we look back, with the map at hand, discerning the motif and further connections and extensions. We will be stressing intuitions rather than formality. Formal statements and outlines of the correctness proofs are collected in Appendix A.

3 The koan

The fundamental tautology of Church numerals is easy to overlook:

$$\mathsf{c}_n \doteq \mathsf{c}_n\,\mathsf{succ}\,\mathsf{c}_0 \qquad (*)$$

That is, the numeral c_n that represents n is the n-times repeated application of the successor succ to c_0. The deep meaning of this triviality unfolds as we go along; Section 8 summarizes why the name "koan" is fitting.

The paper's title promises many predecessors. To conveniently deal with variations without overloading the notation, we introduce "local" definitions $\mathtt{name} :\doteq e$, limited in scope to the section or the explanation block where they appear. The example is immediately below. Locally defined names are set off in a different font from the ordinary, global definitions.

As the first step, (*) gives the recipe for other representations of natural numbers—call them p_n:

$$\mathsf{p}_n \;:\doteq \mathsf{c}_n\,\mathtt{supp}\,\mathsf{p}_0 \qquad n > 0 \qquad (**)$$

This is a definition schema, or a recipe, with supp and p_0 as parameters. Since $\mathsf{c}_0\,\mathtt{supp}\,\mathsf{p}_0 \doteq \mathsf{p}_0$, the parameter p_0 may be regarded as the initial (zeroth) element of the p_n sequence. As to supp, we observe from Equation (1) that:

$$\mathsf{p}_{n+1} \doteq \mathsf{c}_{n+1}\,\mathtt{supp}\,\mathsf{p}_0 \doteq \mathtt{supp}\,(\mathsf{c}_n\,\mathtt{supp}\,\mathsf{p}_0) \doteq \mathtt{supp}\,\mathsf{p}_n$$

That is, $\mathtt{supp}$ acts as the "step function" of the sequence. One may thus say that given the initial element and the step function, (**) is the closed-form expression for the n-th element of the sequence defined by these parameters.

Albeit trivial, the above observations lead to interesting results. For example, instantiating the schema (**) with $\mathtt{p}_0$ as $\mathtt{c}_m$ for some m and $\mathtt{supp}$ as $\mathtt{succ}$ constructs "m-shifted" numerals $\mathtt{p}_0 := \mathtt{c}_m$, $\mathtt{p}_1 := \mathtt{c}_{m+1}$, etc. Since $\mathtt{p}_n$ is $\mathtt{c}_{n+m}$, (**) immediately gives the expression for adding Church numerals: $\mathtt{add} := \lambda nm.n\ \mathtt{succ}\ m$. Kleene's predecessor emerges from the similar, "half-way shifted", numerals, as we see next.

Appendix A reveals that (**) is also the recipe for proving properties of thus constructed $\mathtt{p}_n$ and, ultimately, the correctness of the predecessors.

4 Kleene's predecessor

To obtain the Kleene predecessor, we take as $\mathtt{p}_n$ a point between two consecutive numbers $\mathtt{c}_n$ and $\mathtt{c}_{n-1}$ on the number line. It can be represented as a pair $(\mathtt{c}_{n-1}, \mathtt{c}_n)$:

$$\mathtt{p}_0 := (\mathtt{c}_{-1}\,, \mathtt{c}_0) \qquad \mathtt{p}_1 := (\mathtt{c}_0\,, \mathtt{c}_1) \qquad \mathtt{p}_2 := (\mathtt{c}_1\,, \mathtt{c}_2) \qquad \ldots$$

Here, $\mathtt{c}_{-1}$ is the term that we want as the result of applying the predecessor to $\mathtt{c}_0$—for example, $\mathtt{c}_0$ itself. The successor on those "midpoint numbers" is easy to define

$$\mathtt{supp} :\doteq \lambda p.(\mathsf{snd}\, p, \mathsf{succ}\, (\mathsf{snd}\, p)) \tag{2}$$

With thus chosen $\mathtt{supp}$ and $\mathtt{p}_0$, schema (**) gives the closed-form expression for $\mathtt{p}_n$, from which we can extract $\mathtt{c}_{n-1}$ as the first component:

$$\mathtt{pred} :\doteq \lambda n.\mathsf{fst}\, (n\ \mathtt{supp}\ \mathtt{p}_0) \tag{3}$$

or, in the desugared, normal form:

$$\lambda n.n\ (\lambda ps.s\ (p(\lambda xy.y))\ (\lambda fx.f\ (p(\lambda xy.y)f\ x)))\ (\lambda p.p\ (\lambda fx.x)\ (\lambda fx.x))\ (\lambda xy.x) \tag{4}$$

This is the textbook predecessor (explained, e.g., in the widely used Pierce, 2002). Its size is 41.

5 More predecessors, generally

In Equation (2), $\mathtt{supp}$ receives a pair as the argument but uses only its second component—hinting that something simpler than a pair might do. A simpler representation does come when the "half-shifted" numerals of Section 4 are replaced by "down-shifted". That is, we now take $\mathtt{p}_{n+1}$ to be $\mathtt{c}_n$. We then look for a suitable term $\mathtt{p}_0$ to prepend to this $\mathtt{p}_1\,, \mathtt{p}_2\,, \ldots$ sequence as the initial element. The step function $\mathtt{supp}$ of the resulting sequence usually becomes apparent. Then (**) gives the closed expression for the n-th element of the sequence: $\mathtt{p}_{n+1} :\doteq \mathtt{c}_{n+1}\ \mathtt{supp}\ \mathtt{p}_0$. Recalling that $\mathtt{p}_{n+1}$ is actually $\mathtt{c}_n$ gives what we were looking for: the way to compute $\mathtt{c}_n$ given $\mathtt{c}_{n+1}$.

The general way of extending a sequence X (a set, in general) is embedding it in a longer sequence: the $X\ \mathtt{option}$ construction, which we explore and explain in this section. In contrast, Section 6 extends the set of Church numerals by relying on specific properties of its elements.

X option is the sum data type, with the elements $\{\mathsf{None}\} \cup \{\mathsf{Some}\, x \mid x \in X\}$: the second component of the union embeds X, whereas None is the extra element. Here

$$\mathsf{None} := \lambda k.\lambda y.y \qquad\qquad \mathsf{Some} := \lambda x.\lambda k.\lambda y.kx \tag{5}$$

The downshifted numerals p_n thus become $\mathsf{p}_0 :\doteq \mathsf{None}$, $\mathsf{p}_1 :\doteq \mathsf{Some}\, \mathsf{c}_0$, etc.—in general keeping in mind Equation (5):

$$\mathsf{p}_n\; k\; y \quad \doteq \begin{cases} y & \text{if } n = 0 \\ k\;\; \mathsf{c}_{n-1} & \text{otherwise} \end{cases} \qquad \doteq \begin{cases} y & \text{if } n = 0 \\ k\; (\mathsf{succ}^{(n-1)}\; \mathsf{c}_0) & \text{otherwise} \end{cases} \tag{6}$$

Thus, p_{n+1} are not c_n themselves but their embedding $\mathsf{Some}\, \mathsf{c}_n$, from which one can always project c_n.

The operation supp, to obtain the next p_n in the series, is hence:

$$\mathsf{supp} :\doteq \lambda p.\mathsf{Some}\; (p\; \mathsf{succ}\; \mathsf{c}_0)$$

which gives, via (**), the closed-form expression for p_{n+1}, from which we extract c_n using Equation (6), eventually obtaining the predecessor as:

$$\lambda n.(n\; \mathsf{supp}\; \mathsf{p}_0)\; \mathsf{id}\; \mathsf{c}_0 \tag{7}$$

or, in the desugared, normal form:

$$\lambda n.n\; (\lambda pky.k\; (p(\lambda nfx.f\; (n\, f\; x))(\lambda fx.x)))\; (\lambda ky.y)\; (\lambda x.x)\; (\lambda fx.x) \tag{8}$$

With size 35, it is a bit shorter than the Kleene predecessor.

Equation (6) points to the more economical embedding:

$$\mathsf{p}_{nfx} := \lambda k.\begin{cases} x & \text{if } n = 0 \\ k\; (f^{(n-1)}x) & \text{otherwise} \end{cases} \tag{9}$$

where f and x are some fixed terms: the parameters of the embedding. Clearly, any c_n can be converted to the corresponding p_{n+1}, from which it can be projected back.

The first element and the step function of sequence (9) are thus:

$$\mathsf{p}_{0fx} := \lambda k.x \qquad\qquad \mathsf{supp}_{fx} := \lambda p.\lambda k.k\; (p\, f)$$

which, via the schema (**), gives us the lambda term for p_{nfx} and eventually the predecessor:

$$\lambda n.\lambda fx.(n\;\; \mathsf{supp}_{fx}\;\; \mathsf{p}_{0fx})\; \mathsf{id} \tag{10}$$

or, in the desugared, normal form:

$$\lambda nfx.n\; (\lambda pk.k\; (p\, f))\; (\lambda k.x)\; (\lambda y.y) \tag{11}$$

At size 18, it is the shortest predecessor found so far (less than half the size of Kleene's), and also the fastest, according to the benchmarks of Table 1. It is mentioned, without derivation, explanation or proof, in Barendregt & Barendsen (2000, Theorem 3.14), (and earlier in Barendregt, 1990, Theorem 2.2.14), with a note giving the credit to J. Velmans.[3] An independent derivation appears in Kemp (2007, Section 7.4.1). The exposition in this

[3] It is possible it was derived in Urbanek (1993). However, the author has not been able to locate that paper.

section not only explains the term (which leads to the correctness proof in Appendix A) but also lets one derive such small and fast "predecessors" for other data structures, as we show in Appendix B for binary trees.

6 More predecessors, specifically

In Section 5, we added a new element to Church numerals using the general X option construction that works for any set X: by embedding X into a "bigger" set, which, besides the image $\mathsf{Some}\,X$ also contains the extra element None. In this section,we will be constructing augmented Church numeral sequences by relying on specific properties of the numerals.

As in Section 5, we will be dealing with downshifted numerals $\mathtt{p}_0 := \mathtt{c}_{-1}$, $\mathtt{p}_1 := \mathtt{c}_0$, $\mathtt{p}_2 := \mathtt{c}_1$, etc. This time, however, the sequence $\mathtt{p}_1, \mathtt{p}_2, \ldots$ is not just an embedding of $\mathtt{c}_0, \mathtt{c}_1, \ldots$ but identical to it. The key is to find such a term $\mathtt{c}_{-1}$ that can be easily distinguished from all other Church numerals. We have to make use of some invariant of the numerals.

Here is one invariant: for any $\mathtt{c}_n$, the application $\mathtt{c}_n\ \mathsf{id}$ reduces to id: the identity is a fixpoint of Church numerals. As $\mathtt{c}_{-1}$, we chose a term that, when applied to identity, reduces to something other than the identity, for example, to $\lambda x.\mathtt{c}_0$. The constructors of the $\mathtt{p}_n$ sequence are thus:

$$\mathtt{c}_{-1} := \lambda fx.\mathtt{c}_0 \qquad\qquad \mathsf{supp} :\doteq \lambda p.p\ \ \mathsf{id}\,(\mathsf{succ}\,p)$$

which leads, in the already established route, to the predecessor:

$$\lambda n.n\ (\lambda p.p\ \ \mathsf{id}\,(\mathsf{succ}\,p))\ (\lambda fx.\mathtt{c}_0) \tag{12}$$

Or, in the desugared, normal form:

$$\lambda n.n\ (\lambda p.p\ (\lambda x.x)(\lambda fx.f\ (p\,f\ x)))\ (\lambda fxsz.z) \tag{13}$$

Of size 24, it is nearly half the size of the Kleene predecessor. This was the predecessor that I found in 1992.

Unfortunately, the test that discriminates $\mathtt{c}_{-1}$ from $\mathtt{c}_n$—the application to id—takes linear in n time to reduce. A straightforward modification makes a constant-time test. We do not pursue this approach further (but see the accompanying code). Rather, we demonstrate a different way to look at the augmented Church numerals, taking (*) to the heart. It leads to, arguably, the most inspiring predecessor, with the elegant correctness proof.

As before, the construction is based on (**) with $\mathtt{p}_0$ being $\mathtt{c}_{-1}$ and $\mathtt{supp}$ as mere $\lambda p.p\ \ \mathsf{succ}\,\mathtt{c}_1$, to be called succ°. The predecessor of $\mathtt{c}_n$ is thus the corresponding $\mathtt{p}_n$ itself:

$$\mathtt{pred} := \lambda n.n\ \ \mathsf{succ}^\circ\ \mathtt{c}_{-1} \tag{14}$$

Or, in the desugared, normal form (size 25):

$$\lambda n.n\ (\lambda p.p\ (\lambda cfx.f(cfx))(\lambda x.x))\ (\lambda fxsz.z) \tag{15}$$

The correctness proof is the calculation:

$$\begin{aligned}\mathtt{pred}\,\mathtt{c}_{n+1} &\doteq \mathtt{c}_{n+1}\ \mathsf{succ}^\circ\ \mathtt{c}_{-1} \doteq \mathtt{c}_n\ \mathsf{succ}^\circ\ ((\lambda p.p\ \ \mathsf{succ}\,\mathtt{c}_1)\ (\lambda fx.\mathtt{c}_0))\\ &\doteq \mathtt{c}_n\ \mathsf{succ}^\circ\ \mathtt{c}_0 \doteq \mathtt{c}_n\ \mathsf{succ}\ \mathtt{c}_0 \doteq \mathtt{c}_n\end{aligned}$$

crucially relying on (*). The key step is the fact succ° is itself a successor: $\mathsf{succ}^\circ\ \mathsf{c}_i \doteq \mathsf{c}_i\ \mathsf{succ}\ \mathsf{c}_1 \doteq \mathsf{c}_{i+1}\ \mathsf{succ}\ \mathsf{c}_0 \doteq \mathsf{c}_{i+1}$ for each $i \geq 0$, which again relies on (*), and on Equation (1). Thus, Equation (14) is an extension of (*) with the "metacircular" successor succ°, which behaves just like succ on Church numerals and admits c_{-1} as minus one.

Perhaps the slightly optimized version of Equation (14) (with composition instead of applications) brings up the insight with more force:

$$\lambda n.n\ (\lambda pf.p\ (\mathsf{comp}\, f)f)(\lambda fx.\mathsf{id}) \tag{16}$$

Or, in the desugared, normal form (size 21):

$$\lambda n.n\ (\lambda pf.p\ (\lambda gx.f\ (g\, x))\, f)\ (\lambda fxs.s) \tag{17}$$

7 Un-application

We began by saying that the predecessor is so hard to believe in because it is effectively an un-application. It seems fitting to end by demonstrating it is indeed the case. In fact, we will actually derive the predecessor using un-application.

To be sure, lambda calculus has no un-application rule or operation. We may only apply lambda terms but not examine them. However, lambda calculus can represent, or encode, all computations, including of itself. The representations can be examined and deconstructed to our heart's content. For example, the iterated application $f^{(n)}x$ (for some fixed f and x) may be represented by the already familiar, from Section 5, X option construction: None stands for x and Some e represents the application of f to e:

$$\mathsf{None}_x := \lambda k.x \qquad\qquad \mathsf{Some}_x := \lambda a.\lambda k.k\ a \tag{18}$$

The definitions are parameterized by x, which makes them smaller than those in Equation (5).[4] Thus, $f^{(n)}x$ is encoded as $\mathsf{Some}_x^{(n)}\ \mathsf{None}_x$, which we will call p_{nx} in this section; they are almost the same as p_{nfx} of Equation (9), only with Some_x inplace of f, and without the downshift: p_{nx} corresponds to c_n, whereas p_{nfx} of Equation (9) corresponded to c_{n-1}. From Equation (18), it follows that $\mathsf{p}_{nx}\ \mathsf{id}$ reduces to $\mathsf{p}_{(n-1)x}$ when $n > 0$ and to x otherwise—which is effectively the pattern-matching on p_{nx}.

The construction of p_{nx} from c_n—the encoding, or *reification* (Bawden, 1988; Dybjer & Filinski, 2002) of c_n—is given by (**), or, concretely as:

$$\mathsf{reif}_x := \lambda n.n\ \mathsf{Some}_x\ \mathsf{None}_x \qquad\qquad \mathsf{refl}_f := \mathsf{fix}\ \lambda s.\lambda p.p\ (\lambda q.f\ (s\ q)) \tag{19}$$

The decoding, or *reflection*, recursively interprets p_{nx}, effectively replacing None_x and Some_x with what they are meant to represent: x and the application of f, respectively. Here, fix is the fixpoint combinator. Clearly, $\mathsf{c}_n f\ x \doteq \mathsf{refl}_f\ (\mathsf{reif}_x\ \mathsf{c}_n)$ for any n. Because of fix, refl_f has no normal form, unlike all other terms in this paper. We now rub the blemish away. Contrast the characteristic equality of fix with the consequence of Equation (1):

$$\mathsf{fix}\, e \doteq e\, (\mathsf{fix}\, e) \qquad\qquad \mathsf{c}_{m+1}\ e\ e' \doteq e\, (\mathsf{c}_m\ e\ e') \tag{20}$$

[4] Although Some_x does not include x, we still subscript it to distinguish from Some—and because it is related to None_x, as we see in Section 8.

One may say, c_{m+1} is a "finite" approximation of fix, good up to m recursions. To be precise, if for some e and e_1 the term $\mathsf{fix}\, e\, e_1$ has a normal form, there clearly must exist the number m such that $\mathsf{fix}\, e\, e_1 \doteq (e^{(m)}\ e')\ e_1 \doteq \mathsf{c}_m\, e\, e'\, e_1$, for an *arbitrary* e'.

We hence introduce

$$\mathsf{refl}'_f := \lambda m.m\ (\lambda s.\lambda p.p\ (\lambda q.f\ (s\ q)))\ e' \tag{21}$$

with the property $\mathsf{c}_n\, f\, x \doteq \mathsf{refl}'_f\ \mathsf{c}_m\ (\mathsf{reif}_x\ \mathsf{c}_n)$ for any $m > n$. The number n is being reflected, whereas m drives the reflection. We may even let n itself drive the reflection of its reified predecessor. The term e' truly can be chosen arbitrary; as we will see it only comes to matter when determining the predecessor of c_0, which is generally an open choice. It is simplest to let e' be a bound variable in scope, such as m.

As we have already said, the p_{nx} encoding lets us pattern-match on it and hence remove the outer Some constructor if there was any (otherwise, return x). Hence, the predecessor on p_{nx} numerals is $\mathtt{predp} := \lambda p.p\ \mathsf{id}$. Thus composing reification, predp and reflection gives us the predecessor on Church numerals as:

$$\lambda n.\lambda f x.\mathsf{refl}'_f\ n\ (\mathtt{predp}\ (\mathsf{reif}_x\ n)) \tag{22}$$

Or, in the normal form (size 31):

$$\lambda n.\lambda f x.n\ (\lambda s.\lambda p.p\ (\lambda q.f(sq)))\ n\ (n\ (\lambda a k.ka)\ (\lambda k.x)\ (\lambda z.z)) \tag{23}$$

One might think that with the piling up of reflection onto reification, the result would be awful. Yet predecessor (23) is smaller and faster than the Kleene predecessor (4)—in some cases, one of the fastest, as we see next.

8 Connections

Let us look back and draw a map, to help in further travel. The seemingly quasi-random wanderings have all been the variations of the same motif, about encodings, data types, and algebras (with the operations c_0 of arity 0 and succ of arity 1.) In particular, everything seems to revolve around the functor $F(X) := 1 + X$. Church numerals and the algebraic data type `type nat = Succ of nat | Zero` are the carrier sets of two (isomorphic, by definition) initial F-algebras for this functor. Then (**) expresses the unique homomorphism, from the initial algebra of Church numerals to the algebra with the carrier set p_n.

The functor $F(X)$ represents the data type X option; its fixpoint, $\mu X.(X\ \mathsf{option})$, is none other than nat. This is the idea behind Equation (19) in Section 7. We have used two encoding of the X option data type: Böhm & Berarducci (1985) in Equation (5) and Scott–Mogensen (Mogensen, 1992; Abadi *et al.*, 1993) in Equation (18).

The seemingly trivial (*) appears by the same name in Böhm & Berarducci (1985). One understands its significance only when rediscovers it for oneself—as it happened to Wadler[5] and the author.[6]

[5] http://www.seas.upenn.edu/~sweirich/types/archive/1999-2003/msg00138.html.
[6] http://okmij.org/ftp/tagless-final/course/Boehm-Berarducci.html.

Table 1. *Size and performance comparison of various predecessors. Equation* (4) *is the original Kleene predecessor. The third column shows the number of normal-order reductions to normalize* $\mathtt{pred}\,\mathtt{c}_{100}$. *The normalization of Equation* (13) *did not finish in 5 minutes; the shown number is obtained by extrapolation. The last two columns show the performance metrics (using the built-in* `time`*) of evaluating* `(((pred c10000) incr) 0)` *on Petite Chez Scheme Version 8.4 on AMD64. Here,* `incr` *is defined as* `(lambda (n) (+ 1 n))`

Predecessor	Size	Reductions to normalize $\mathtt{pred}\,\mathtt{c}_{100}$	`(((pred c10000) incr) 0)` time (ms)	memory (MB)
Equation (4)	41	604	1281	3207
Equation (8)	35	603	4	2.1
Equation (11)	18	205	3	0.6
Equation (13)	24	$> 4 * 2^{100}$	1348	3127
Equation (15)	25	14757	1651	4675
Equation (17)	21	9906	1312	3128
Equation (23)	31	506	5	1.5

The $\mathtt{p}_n$ number representation, completely specified per (**) by $\mathtt{p}_0$ and `supp`, is called "numeral system" in Barendregt (1981, Section 6.4) (Numeral systems are required to also possess the zero-test operation, which is not needed for our development. The exercises to its Section 6 discuss other numeral systems, including binary.) Barendregt (1981) introduces one particular $\mathtt{p}_n$, denoted $\ulcorner n \urcorner$ in his book (Definition 6.2.9), with the straightforward predecessor, and the isomorphism to $\mathtt{c}_n$ witnessed by lambda terms. Therefore, the predecessor on $\ulcorner n \urcorner$ can be "conjugated" to give the predecessor on Church numerals (Corollary 6.4.6). This is the essence of the approach we exposed in Section 7. The requirement that the isomorphism between $\mathtt{p}_n$ and $\mathtt{c}_n$ be witnessed by lambda terms is, however, too strong: Section 7 gets by without it. Its refl_f and reif_x express only a part of the isomorphism, and their composition is not the identity. As another difference, refl'_f does not use the fixpoint combinator and hence has a normal form. All our predecessors have normal form.

Table 1 compares the predecessors. Although the performance of lambda calculus predecessors is not something one would lose sleep over (except for the author), we evaluate it as well, as the number of normal-order reductions to normalize $\mathtt{pred}\,\mathtt{c}_{100}$ (giving $\mathtt{c}_{99}$). These numbers in the table are computed by the embedding of lambda calculus in OCaml; the complete code, with more examples, is available at `http://okmij.org/ftp/tagless-final/pred.ml`. One should keep in mind that the normal reduction strategy substitutes expressions that may have redices, with the ever-present danger of exponential explosion (which indeed occurs in the case of Equation (13)). As a more realistic test, we show the time and memory it takes to evaluate `(pred c10000)` and then to convert it to an integer, on Petite Chez Scheme, a highly optimizing Scheme compiler. All performance tests used the normal form of the predecessors.

Thus, looking back, the overarching idea has been the construction of an initial algebra for the $F(X)$ functor. Although isomorphic to the $\mathtt{c}_n$ initial algebra, it is designed to have an easily expressible predecessor. In the light of F-algebras, the general approaches in Sections 5 and 7 now look systematic: The $X\,\mathtt{option}$ construction was not arbitrary; it was the representation of the $F(X)$ functor in question. The general predecessor approaches thus extend to the Church encoding of any other algebraic data type (initial

F-algebra)—*mechanically*: write down the functor, write down the corresponding data type construction, apply Böhm–Berarducci or Scott–Mogensen encoding following the steps of Sections 5 and 7, and obtain an efficient predecessor/extractor. Appendix B illustrates, for binary trees.

The specific approach in Section 6, by its nature, does not generalize so easily. Still, the predecessors in Section 6, although not particularly useful for anything, are pleasing to the eye and to the mind—like a real pearl.

9 Conclusions

> Our reality may be very much like theirs. All this might just be an elaborate simulation running inside a little device sitting on someone's table.
> StarTrek TNG, Episode 6x12, "Ship in a Bottle"

The tricky predecessor turned prosaic, once we have changed the point of view—which came about from contemplating representations and what they represent. The metacircular successor in Equation (14) is the case point, of the epigraph as well. With what we know now about algebraic data types and their representations, the predecessor is no longer a mystical term requiring alternative states of mind and tooth sacrifices. We have also experienced the excitement of revisiting the Canon—and the wonder at the delicate behavior that arises from trite rules.

Acknowledgments

I thank Peter Hancock for inspiring discussions. The anonymous referees have pushed to polish the pearl harder and gave many helpful suggestions. This work was partially supported by JSPS KAKENHI Grant Number 17K00091.

Conflict of interest

There is no conflict of interest to declare.

Supplementary materials

To view supplementary material for this article, please visit http://doi.org/10.1017/S095679682000009X.

References

Abadi, M., Cardelli, L. & Plotkin, G. D. (1993) Types for the Scott numerals. Available at: http://lucacardelli.name/Papers/Notes/scott2.pdf.

Barendregt, H. (1997) The impact of the lambda calculus in logic and computer science. *Bull. Symb. Log.* **3**(2), 181–215.

Barendregt, H. & Barendsen, E. (2000) *Introduction to Lambda Calculus*.

Barendregt, H. P. (1981) *The Lambda Calculus: Its Syntax and Semantics*. Amsterdam: Elsevier.

Barendregt, H. P. (1990) Functional programming and lambda calculus. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, van Leeuwen, J. (ed), Elsevier and MIT, pp. 321–363.
Bawden, A. (1988) Reification without evaluation. Memo 946. Artificial Intelligence Laboratory, Massachusetts Institute of Technology.
Böhm, C. & Berarducci, A. (1985) Automatic synthesis of typed Λ-programs on term algebras. *Theor. Comput. Sci.* **39**(2–3), 135–154.
Dybjer, P. & Filinski, A. (2002) Normalization and partial evaluation. In *APPSEM 2000: International Summer School on Applied Semantics, Advanced Lectures*, Barthe, G., Dybjer, P., Pinto, L. & Saraiva, J. (eds). Lecture Notes in Computer Science, vol. 2395. Springer, pp. 137–192.
Kemp, C. J. M. (2007) *Theoretical Foundations for Practical 'Totally Functional Programming'*. Ph.D. thesis, University of Queensland.
Mogensen, T. Æ. (1992) Efficient self-interpretation in lambda calculus. *J. Funct. Program.* **2**(3), 345–364.
Pierce, B. C. (2002) *Types and Programming Languages*. MIT.
Statman, R. (1979) The typed λ-calculus is not elementary recursive. *Theor. Comput. Sci.* **9**(1), 73–81.
Urbanek, F. J. (1993) A simple lamda-term representing the predecessor function with respect to church numerials. *Bull. EATCS* **50**, 276–279.
Whitehead, A. N. & Russell, B. (1910). *Principia Mathematica, Volume I*. Cambridge, UK: Cambridge University.

A Appendix

Correctness formalities

The correctness of a predecessor term $\mathtt{pred}$ is expressed by the following property:

$$\mathtt{pred}\,\mathsf{c}_{n+1} \doteq \mathsf{c}_n \qquad \forall n \geq 0 \tag{A1}$$

which also states that $\mathtt{pred}\,\mathsf{c}_{n+1}$ has the normal form, viz. c_n, and hence can be reduced to it with the normal reduction strategy. This section outlines the proofs of this property for the predecessors introduced in this paper.

The proofs are centered around three basic properties of Church numerals: Equation (1), (*) and the following: Assume f, h, and g are the terms such that $\mathsf{h} \circ \mathsf{f} \doteq \mathsf{g} \circ \mathsf{h}$. Then

$$\mathsf{h} \circ (\mathsf{c}_n\,\mathsf{f}) \doteq (\mathsf{c}_n\,\mathsf{g}) \circ \mathsf{h} \qquad \forall n \geq 0 \tag{A2}$$

Intuitively, if one can "push" h past one application of f, one can push it past any number of the consecutive applications of f. These three properties can be demonstrated by straightforward induction, or algebraically. On the other hand, induction is not needed for the correctness proofs themselves, below. The proofs are based on equational re-writing and are calculational in nature.

The general way of constructing a predecessor is given c_{n+1}, first build the term p_{n+1} using (**) with the appropriate $\mathtt{supp}$ and p_0. By construction, it should be easy to extract c_n from p_{n+1}; we call the extraction term $\mathtt{rfl}$. All in all, we have the following construction schema:

$$\mathtt{pred} :\doteq \lambda n.\mathtt{rfl}\,(n\;\mathtt{supp}\;\mathsf{p}_0) \tag{A3}$$

Suppose the following two conditions hold

$$\mathtt{h} \circ \mathtt{supp} \doteq \mathsf{succ} \circ \mathtt{h} \qquad \mathtt{h}\,\mathsf{p}_0 \doteq \mathsf{c}_0 \qquad \text{where } \mathtt{h} :\doteq \mathtt{refl} \circ \mathtt{supp} \tag{A4}$$

Then, by simple equational reasoning, using Equations (1) and (A2) and (*):

$$\begin{aligned}\mathtt{pred}\,\mathsf{c}_{n+1} &\doteq \mathtt{rfl}\,(\mathsf{c}_{n+1}\,\mathtt{supp}\,\mathsf{p}_0) \doteq \mathtt{rfl}\,(\mathtt{supp}\,(\mathsf{c}_n\,\mathtt{supp}\,\mathsf{p}_0)) \doteq \mathtt{h}\,(\mathsf{c}_n\,\mathtt{supp}\,\mathsf{p}_0)\\ &\doteq \mathsf{c}_n\,\mathsf{succ}\,(\mathtt{h}\,\mathsf{p}_0) \doteq \mathsf{c}_n\,\mathsf{succ}\,\mathsf{c}_0 \doteq \mathsf{c}_n\end{aligned}$$

That is, provided Equation (A4) hold for the chosen supp and p_0, the predecessor constructed according to schema (A3) is correct.

Proving the correctness of a predecessor thus amounts to checking the conditions (A4). For example, for Kleene's predecessor (4), p_0 is $(\mathsf{c}_{-1}, \mathsf{c}_0)$, supp is $\lambda p.(\mathsf{snd}\,p, \mathsf{succ}\,(\mathsf{snd}\,p))$ and rfl is fst. Then $\mathtt{h} :\doteq \mathtt{rfl} \circ \mathtt{supp} \doteq \mathsf{snd}$. It is easy to see that $\mathtt{h}\,\mathsf{p}_0$ reduces to c_0 and $\mathtt{h} \circ \mathtt{supp}$ indeed equals to $\mathsf{succ} \circ \mathtt{h}$ by doing a couple of substitutions in one's head (or normalizing both terms and comparing the results—the approach taken in the accompanying code). The Kleene predecessor is indeed correct. The correctness of Equations (8) and (11) can be seen just as mechanically.

Predecessors derived in the "specific" way, in Section 6, have specific, and simpler correctness proofs. Recall, the specific construction schema for the predecessors is

$$\mathtt{pred} :\doteq \lambda n.n\,\mathtt{supp}\,\mathsf{p}_0 \tag{A5}$$

where supp and p_0 are chosen so that the following holds

$$\mathtt{supp}\,\mathsf{p}_0 \doteq \mathsf{c}_0 \qquad \mathtt{supp}\,\mathsf{c}_n \doteq \mathsf{succ}\,\mathsf{c}_n \qquad \forall n \geq 0 \tag{A6}$$

These conditions indeed guarantee the correctness:

$$\mathtt{pred}\,\mathsf{c}_{n+1} \doteq \mathsf{c}_{n+1}\,\mathtt{supp}\,\mathsf{p}_0 \doteq \mathsf{c}_n\,\mathtt{supp}\,(\mathtt{supp}\,\mathsf{p}_0) \doteq \mathsf{c}_n\,\mathtt{supp}\,\mathsf{c}_0 \doteq \mathsf{c}_n\,\mathsf{succ}\,\mathsf{c}_0 \doteq \mathsf{c}_n$$

using Equation (1) and (*). That these conditions hold for Equation (14) is shown in Section 6; for the others in that Section, the checks are just as straightforward.

The correctness of Equation (23) depends, foremost, on the correctness of reflection/reification:

$$\lambda fx.\mathsf{refl}_f\,(\mathsf{reif}_x\,\mathsf{c}_n) \doteq \mathsf{c}_n \qquad \forall n \geq 0 \tag{A7}$$

It is easy to check by calculation that $\mathsf{refl}_f \circ \mathsf{Some}_x \doteq f \circ \mathsf{refl}_f$. Then Equation (A7) immediately follows from Equation (A2). Furthermore, the reduction of $\mathsf{refl}_f\,(\mathsf{Some}_x^{(n)}\,\mathsf{None}_x)$ to $f^{(n)}x$ requires performing of no more than $n+1$ reductions of the sort $\mathsf{fix}\,e$ to $e(\mathsf{fix}\,e)$ ("unrolling of the fixpoint"). This justifies the replacement of refl_f with $\mathsf{refl}'_f\,\mathsf{c}_{n+1}$ in the above refl_f reduction.

B Appendix

Predecessors on trees

The general approaches for constructing a predecessor of Church numerals in Sections 5 and 7 are general indeed and apply to the Church encoding of any algebraic data type (initial F-algebra). As an illustration, this section uses them to build an extractor of a

branch from a binary tree. The accompanying code contains the complete development, closely following the explanations in the paper; the following describes its salient points.

Binary trees with leaves containing data from some set A are described by the functor $F_A(X) := A + X \times X$, or, in a programming language notation

```
type ('a,'x) tree = Leaf of 'a | Node of 'x * 'x
```

The corresponding Church initial algebra has operations leaf of arity 0 (but with the parameter A) and node of arity 2, defined as follows:

$$\mathsf{leaf} := \lambda a.\lambda fg.f\ a \qquad\qquad \mathsf{node} := \lambda t_1 t_2.\lambda fg.g(t_1 fg)\,(t_2 fg)$$

We use t as a metavariable for a Church-encoded tree. Any such tree is constructable using the operations of the algebra: $t \doteq t\ \mathsf{leaf}\ \mathsf{node}$, which is the analogue of (*). The goal is to find branch extractors terms `left` and `right` with the following property:

$$\texttt{left}\ (\mathsf{node}\ t_1\ t_2) \doteq t_1 \qquad\qquad \texttt{right}\ (\mathsf{node}\ t_1\ t_2) \doteq t_2 \qquad \text{for any trees } t_1, t_2$$

Given any other tree algebra (whose operations we will be calling `pleaf` and `pnode`), the (unique) homomorphism from the Church initial algebra is computed by $t \mapsto t\ \texttt{pleaf}\ \texttt{pnode}$. This is the analogue of (**).

The reflection-reification approach of Section 7 relies on the (optimized) Scott–Mogensen encoding of the algebraic data type that is a carrier of the initial F_A-algebra:

$$\texttt{Leaf}_f := \lambda a.\lambda g.f\ a \qquad\qquad \texttt{Node}_f := \lambda t_1 t_2.\lambda g.g\ t_1\ t_2$$

The reification and reflection perform conversions:

$$\texttt{reif}_f \doteq \lambda t.t\ \texttt{Leaf}_f\ \texttt{Node}_f \qquad\qquad \texttt{refl}_g \doteq \mathsf{fix}\ \lambda s.\lambda t.t(\lambda t_1 t_2.g(st_1)(st_2))$$

Extracting the left and the right branch from $\texttt{Node}_f\ p_1\ p_2$ cannot be simpler: $\texttt{pleft} := \lambda p.p\ \mathsf{true}$ and similarly for `pright`. Thus, the left and right extractors for Church-encoded trees are obtained by converting a tree to the Scott–Mogensen encoding, extracting the branch there, and reflecting it back to the Church encoding:

$$\texttt{left} \doteq \lambda t.\lambda fg.\texttt{refl}_g\ (\texttt{pleft}\ (\texttt{reif}_f\ t))$$

As in Section 7, fix can be avoided, by letting the tree drive its own reflection. Obtaining `left` and `right` that have a normal form is left as an exercise (the accompanying code shows the answer).

Section 5, in contrast, relies on the Böhm–Berarducci encoding of `tree`; here it is, in the optimized form as explained in the second half of Section 5:

$$\texttt{pleaf}_f := \lambda a.\lambda k.f\ a \qquad\qquad \texttt{pnode}_g := \lambda t_1 t_2.\lambda k.k\ (t_1 g)\ (t_2 g)$$

Then the `left` extractor is obtained as $\lambda t.\lambda fg.t\ \texttt{pleaf}_f\ \texttt{pnode}_g\ \mathsf{true}$—or, in the normal form:

$$\lambda t.\lambda fg.t\ (\lambda a.\lambda k.fa)\ (\lambda t_1 t_2.\lambda k.k\ (t_1 g)\ (t_2 g))\ (\lambda xy.x)$$