Lambda Calculus

Michael Weiss

November 7, 2018

1 Introduction

These are based on some notes I wrote up in the early 1970s, for a seminar on an early version of Dana Scott's model of the *untyped* lambda calculus. These notes have nothing to say about Scott's subsequent work on the typed lambda calculus, and the general notion of types. Also, it seems that Plotkin independently came up with the same ideas a little before Scott.

$\lambda x.[\ldots x \ldots]$ is the function $x \mapsto [\ldots x \ldots]$. E.g.: $\lambda x.x^2 + 1$ is the function taking x to $x^2 + 1$.

If f is a function and x is an argument, for $f(x)$ we write simply fx.

We will now push the λ notation a little further than at first seems to make sense.

Some examples: set $I = \lambda x.x$, the identity function. So $II = I$, because for all x, $Ix = x$. Next, set

$$K = \lambda x.(\lambda y.x)$$

So Kx is the constant function taking everything to x. $(KK)I = K$. Fur-

thermore, for all x, y, and z,

$$(([K(KI)]x)y)z = ((KI)y)z = Iz = z$$

so if we think of $K(KI)$ as a function of three variables, it is a projection function: $\langle x, y, z\rangle \mapsto z$. Next, set

$$D = \lambda x.xx$$

So $DI = II = I$. $DK = KK$, the constant function taking everything to K. $DD = DD$—the definition of D leads to a circle when applied to D.

Because of the last difficulty, as well as a reluctance to allow a function to be its own argument (even in such cases as II), we change our interpretation. Think of λ-terms as representing, not functions, but instructions for computing functions (i.e., "programs"). They describe procedures.

These procedures operate on data of some sort, and the programs themselves are an admissible type of data. A procedure might not terminate for a given argument, so we get partial functions.

2 A model of lambda-calculus

Let $\{\varphi_i\}$ be an enumeration of the partial recursive functions of one variable. (We say that i is an index for φ_i.) There is a 1–1 correspondence between indices and programs: $i \leftrightarrow P_i$, where P_i is a program that computes φ_i. Because the correspondence is 1–1, we might as well think of P_i as *being* i. So program i applied to input j yields $\varphi_i(j)$. Formally,

$$ij = \varphi_i(j)$$

(Of course, ij is not always defined.) Interpret $\lambda x.[\ldots x \ldots]$ as follows: if $x \mapsto [\ldots x \ldots]$ is a partial recursive function, and we choose an index i for

it in some canonical way (so we have $\varphi_i(x) = [\ldots x \ldots]$ for all x), then let $\lambda x.[\ldots x \ldots]$ be that index i. E.g., $\lambda x.xx$ is an index for the partial recursive function $x \mapsto \varphi_x(x)$, so if $d = \lambda x.xx$, then

$$(\forall x)(\varphi_d(x) = \varphi_x(x))$$

The equal-sign means the one side is defined iff the other side is, in which case both sides are equal. So $\varphi_d(d)$ does not converge. (To show this, it would be necessary to discuss more exactly how d is "canonically" chosen.)

In case the issue of making a canonical choice is bothering you, we will revisit it at the end of the next section. For now, treat this section as motivational.

In general, for a model of the lambda-calculus, we need:

1. a domain $\mathcal{D}$
2. a partial function, called *application*, from $\mathcal{D} \times \mathcal{D}$ to $\mathcal{D}$. This means it should be possible to regard the elements of $\mathcal{D}$ both as "programs" and as "data". The programs define procedures that operate on $\mathcal{D}$; thus in xx, program x takes data x as input.
3. If a function $x \mapsto [\ldots x \ldots]$ is reasonably "programmable", then there should be an element of $\mathcal{D}$, denoted $\lambda x.[\ldots x \ldots]$, which when applied to x yields $[\ldots x \ldots]$.

3 Lambda language

We make all this more formal. We define a lambda language. All lambda languages are the same, except possibly for some constants, denoting predefined functions. (E.g., you might want to include a a constant for the successor function on the natural numbers. Terminology: for convenience,

"number" will mean "natural number" unless otherwise specified.) The definition follows the usual recursive paradigm:

1. Vocabulary: set of constants (possibly empty), denumerable infinity of variables, and: ( ) . λ. (I.e., parens, dot, and λ.)

2. Definition of a lambda term:

   (a) constants and variables are lambda terms.

   (b) if σ, τ are lambda terms, then so is $(\sigma\tau)$.

   (c) if σ is a lambda term and x is a variable, then $(\lambda x.\sigma)$ is a lambda term.

3. Define free and bound variables and closed lambda terms in the usual way. Notation: $\sigma[x]$ is a lambda term with free variable x; if x is replaced throughout by τ, call the result $\sigma[\tau]$.

4. Conversion: $(\lambda x.\sigma[x])\tau$ converts to $\sigma[\tau]$. If some subterm of σ (perhaps all of σ) is converted, changing σ into σ', we say $\sigma \xrightarrow{(1)} \sigma'$. If

   $$\sigma_0 \xrightarrow{(1)} \sigma_1 \xrightarrow{(1)} \cdots \xrightarrow{(1)} \sigma_k$$

   we say $\sigma_0 \to \sigma_k$. We read this, "σ_0 reduces to σ_k".

5. If σ cannot be reduced, we say σ is normal. If $\sigma \to \tau$ and τ is normal, we say σ is normalizable. E.g.: $(\lambda x.xx)(\lambda x.xx)$ is closed but not normalizable. If $\sigma \to \tau$, τ normal, we say τ is a normal form for σ.

   The Church-Rosser theorem says that normal forms are unique. The proof depends on the Church-Rosser property, which we state without proof: if $\psi \xrightarrow{(1)} \psi'$, and $\psi \xrightarrow{(1)} \psi''$, then there exists a ψ''' such that:

   $$\begin{array}{ccc} \psi & \xrightarrow{(1)} & \psi' \\ {\scriptstyle (1)}\downarrow & & \downarrow \\ \psi'' & \longrightarrow & \psi''' \end{array}$$

Use $\dashrightarrow$ to indicate a partial function (e.g., $f : \mathbb{N} \dashrightarrow \mathbb{N}$), and $f \equiv g$ to mean, for partial f and g, that for all x, $f(x)$ is defined iff $g(x)$ is defined, in which case $f(x) = g(x)$.

Because lambda terms are defined inductively, we can give proofs by induction on the structure of lambda terms. I.e., first prove the desired result for a variable x, then for $(\sigma\tau)$ assuming it's true for σ and τ, and finally for $\lambda x.\sigma[x]$ assuming it's true for $\sigma[x]$. We'll say that we're inducting on the *complexity* of lambda terms.

A *model* for a lambda language consists of a *domain* $\mathcal{D}$, a partial function $\mathcal{D}\times\mathcal{D} \dashrightarrow \mathcal{D}$ (called *application*), and a partial function $\mathcal{V} : \{\text{closed terms}\} \dashrightarrow \mathcal{D}$ (called *interpretation*), such that for all σ,τ:

1. $\mathcal{V}(\sigma\tau) \equiv \mathcal{V}(\sigma)\mathcal{V}(\tau)$ ($\mathcal{V}(\sigma)$ applied to $\mathcal{V}(\tau)$)
2. $\mathcal{V}((\lambda x.\sigma[x])\tau) \equiv \mathcal{V}(\sigma[\tau])$
3. $\mathcal{V}(\sigma)$ is defined for every closed normalizable term σ.

(Strictly speaking we should have added something about substitution equivalence, e.g., $\lambda x.x$ is "really the same as" $\lambda y.y$.)

Let us now reconsider our model from the previous section. We assumed we had some canonical enumeration of programs P_i. For example, P_i might be a description of the i-th Turing machine, or the i-th program in some computer language. Strictly speaking, the enumeration needs to be *effective*, in that the process of going from i to P_i (and vice versa) is completely mechanical. It wouldn't do, for example, to say each P_i must halt on all inputs, or to insist that P_i and P_j compute different partial functions for $i \neq j$. Determining whether a program describes a total function, or whether two programs determine the same partial function, are famously *uncomputable* (i.e., undecidable) problems.

Here's a sketchy description of an effective enumeration, just to be concrete. We start with descriptions of Turing machines; these must adhere to some precise format, so we can list, in lexicographic order, all valid specifications. We let P_i be the i-th such specification. We also give a rule by which a Turing machine defines a partial function $\mathbb{N} \dashrightarrow \mathbb{N}$. Traditionally this is done by writing a string of m 1's on the tape as input, and declaring that the program has computed output n when and only when it halts with a string of n 1's on the tape. If it doesn't halt, or doesn't halt with properly formatted output, the function is undefined for that input.

It's a similar story for the lambda calculus. We can list the closed lambda terms in lexicographic order. Let τ_i be the i-th closed lambda term. Traditionally only *certain* closed lambda terms (so-called Church numerals) are used to represent numbers, but for our purposes, it will be slicker to just say that τ_i represents the number i. So now here is a model of the lambda calculus.

1. The domain $\mathcal{D}$ is $\mathbb{N}$.
2. Application $\langle i, j \rangle \mapsto ij$ is defined thus: if $(\tau_i \tau_j)$ reduces to normal form τ_k, then we let ij equal k, but if $(\tau_i \tau_j)$ is not normalizable, ij is undefined.
3. $\mathcal{V}(\tau_i) = k$ if τ_i reduces to normal form τ_k, but if τ_i is not normalizable, $\mathcal{V}(\tau_i)$ is undefined.

Clearly the Church-Rosser theorem is the linchpin that makes this work. We will call this the *index model* of the lambda calculus.

We can use the index model to define an effective enumeration of the partial recursive functions. Just let

$$\varphi_i^{\lambda}(j) = ij$$

for all i and j, where we have application on the right-hand side. If φ_i^T is the enumeration obtained from Turing machines, then we can go back and forth mechanically between them:

$$\varphi^{\lambda}_{f(i)} \equiv \varphi^T_i$$
$$\varphi^{T}_{g(i)} \equiv \varphi^\lambda_i$$

where the functions f and g are total recursive. All the various definitions for "computable function" that were proposed during the 1930s turned out to be equivalent in this sense.

Lots of minor variations on the index model are possible. Maybe the simplest is just to let the domain $\mathcal{D}$ be the set of closed lambda terms, let application send $\langle \sigma, \tau \rangle$ to the reduced form of $(\sigma\tau)$, if it exists, and let $\mathcal{V}(\sigma)$ be the reduced form of σ, if it exists. To use Church numerals for data, a slight modification of notion of model is needed. So far we've had an empty set of constants, i.e., no predefined functions, but you could add symbols for (say) the constantly 0 function and the successor function on $\mathbb{N}$, and use that to provide a "more natural" definition of the partial recursive functions $\mathbb{N} \dashrightarrow \mathbb{N}$.

All these variations share one feature: the interpretation function $\mathcal{V}$ is not defined on *all* closed terms, just normalizable ones. Scott's model overcame this drawback.

But before we can get to that, we'll need to discuss enumeration operators.

4 Enumeration operators

You can find the formal definition of *enumeration operator* in Rogers, *Theory of Recursive Functions and Effective Computability*, §9.7. I'll build up to the concept, via examples, before presenting the definition.

The important point about enumeration operators is that they have unique minimal fixed points, which are "solutions" to recursive definitions. E.g.: suppose we've defined the set Even $= \{n \in \mathbb{N} | n \text{ is even}\}$ and the function $\lambda n.\frac{n}{2}$ sending Even $\to \mathbb{N}$. Then we define the set Pow2 of powers of 2 by the recursive definition

$$n \in \text{Pow2} \Leftrightarrow [n = 1 \vee (n \in \text{Even} \wedge \frac{n}{2} \in \text{Pow2}]$$

We resolve the circularity by defining an operator $\Phi : \mathcal{P}\mathbb{N} \to \mathcal{P}\mathbb{N}$ as follows:

$$(\forall n \in \mathbb{N}, x \in \mathcal{P}\mathbb{N}) \left[n \in \Phi(x) \Leftrightarrow [n = 1 \vee (n \in \text{Even} \wedge \frac{n}{2} \in x]\right]$$

Then Pow2 is the unique fixed point of Φ, given by

$$\text{Pow2} = \bigcup_{k \in \mathbb{N}} \Phi^k(\varnothing)$$

In fact, $\Phi^k(\varnothing) = \{1, 2, \ldots, 2^{k-1}\}$. Viewed dynamically, Φ creates $\Phi(x)$ by throwing the following into a bag: (1) the number 1, and (2) $2m$ for every $m \in x$.

Second example: $x \subseteq \mathbb{N}$ is closed under multiplication iff

$$(\forall m, n \in \mathbb{N})(m, n \in x \Rightarrow m \cdot n \in x)$$

The multiplicative closure of x is the smallest $y \supseteq x$ that is closed under multiplication. Define $\Psi(x) \in \mathcal{P}\mathbb{N}$ by this condition: for all $n \in \mathbb{N}$ by:

$$n \in \Psi(x) \Leftrightarrow [n \in x \vee (\exists k, l)(n = k \cdot l \wedge k, l \in x)]$$

Then x is closed under multiplication iff $\Psi(x) = x$. The multiplicative closure of x is $\bigcup_{k \in \mathbb{N}} \Psi^k(x)$, the unique minimal fixed point containing x. (If you define $\Psi_x(y) = x \cup \Psi(y)$, then the multiplicative closure of x is the unique minimal fixed point of Ψ_x.)

Consider next a recursive definition of a function, say

$$f(n) = \begin{cases} 1 \text{ if } n = 0 \\ n \cdot f(n-1) \text{ if } n > 0 \end{cases}$$

This time define an operator Θ taking partial functions to partial functions by

$$(\Theta f)(n) = \begin{cases} 1 \text{ if } n = 0 \\ n \cdot f(n-1) \text{ if } n > 0 \end{cases}$$

Of course the factorial function is the unique fixed point of Θ. We may regard partial functions as sets of ordered pairs, or if we use a pairing function $\mathbb{N} \times \mathbb{N} \leftrightarrow \mathbb{N}$, as subsets of $\mathbb{N}$. Let $\langle x, y \rangle$ be a true ordered pair and (m, n) its coding, so $(m, n) \in \mathbb{N}$. Then $f' \in \mathcal{P}\mathbb{N}$ codes a partial function iff $(m, n_1) \in f'$ and $(m, n_2) \in f'$ implies that $n_1 = n_2$, i.e., f' is single-valued. Define $\Theta' : \mathcal{P}\mathbb{N} \to \mathcal{P}\mathbb{N}$ by

$$(m, n) \in (\Theta' f') \Leftrightarrow \begin{cases} m = 0 \wedge n = 1, \vee \\ m > 0 \wedge n = m \cdot l \wedge (m-1, l) \in f' \end{cases}$$

Then the (coded) graph of the factorial functions, $\{(m, n) | n = m!\}$, is the unique fixed point of Θ'. Looked at dynamically, Θ' construes its argument $f' \in \mathcal{P}\mathbb{N}$ as a set of pairs of numbers; it puts (0,1) into the output, and if $(m-1, l) \in f'$, i.e., $f(m-1) = l$, then it puts $(m, m \cdot l) = (m, m \cdot f(m-1))$ into the output. So $(\Theta')^k(\varnothing)$ "computes" $0! = 1$, $1! = 1 \cdot 0!$, up to $(k-1)! = (k-1)(k-2)!$, but $k!$ isn't computed until we get to $(\Theta')^{k+1}(\varnothing)$.

In general, we gloss over the distinction between f and f' and Θ and Θ'. Note that the graph of a partial recursive function f, $\{(m, f(m)) | m \in \mathbb{N}\}$, is a recursively enumerable set, and the graph of a total recursive function is a recursive set (since $(m, n) \notin f'$ iff $(m, k) \in f'$ for some $k \neq n$).

Here's an interesting example of an enumeration operator from Rogers (exercise 11-41, due to Kreisel). (It's a bit of a digression, skip it if you want.)

Define

$$f(x,y) = \begin{cases} 0 \text{ if } \varphi_x(x) \text{ diverges} \\ 2^k \text{ if } \varphi_x(x) \text{ converges in exactly } y+k \text{ steps} \\ 1 \text{ if } \varphi_x(x) \text{ converges in less than } y \text{ steps} \end{cases}$$

(The more steps yet to go, the bigger $f(x,y)$, supposing $\varphi_x(x)$ converges.) Now $f(x,y)$ is not recursive, because $\{x|\varphi_x(x) \text{ diverges}\}$ is not recursive, or even recursively enumerable. But $f(x,y)$ is an extension of a partial recursive f_0:

$$f_0(x,y) = \begin{cases} f(x,y) \text{ if } \varphi_x(x) \text{ converges} \\ \text{undefined if } \varphi_x(x) \text{ diverges} \end{cases}$$

Note that $f(x,y)$ satisfies

$$f(x,y) = \begin{cases} 2f(x,y+1) \text{ if } \varphi_x(x) \text{ does not converge in } \leq y \text{ steps} \\ 1 \text{ if } \varphi_x(x) \text{ converges in } \leq y \text{ steps} \end{cases}$$

For if $\varphi_x(x)$ converges, then $\lambda y.f(x,y)$ looks like

$$2^l, 2^{l-1}, \ldots 2, 1, 1, 1, \ldots$$

where the first 1 appears when y is the number of steps to converge.

Construct an enumeration operator Γ:

$$(\Gamma f)(x,y) = \begin{cases} 2f(x,y+1) \text{ if } \varphi_x(x) \text{ does not converge in } \leq y \text{ steps} \\ 1 \text{ if } \varphi_x(x) \text{ converges in } \leq y \text{ steps} \end{cases}$$

Viewed dynamically, Γ has instructions to put $(x,y,1)$ in the output whenever $\varphi_x(x)$ converges in $\leq y$ steps. For each (x,y,z) such that $\varphi_x(x)$ does not converge in $\leq y$ steps, Γ has these instructions: if $(x,y+1,z)$ is found in the input (i.e., $f(x,y+1) = z$), then put $(x,y,2z)$ in the output (i.e.,

make $(\Gamma f)(x,y) = 2f(x,y+1))$. The instructions for Γ are a recursively enumerable set.

OK, here's the point of this example: f_0 is the unique *minimal* fixed point of Γ, and f is the unique *total* fixed point of Γ. This example shows (stated casually) that set of recursion equations can determine a total, non-partial-recursive function uniquely as its only total solution, even though the unique minimal solution is always partial recursive.

All our examples have this in common: (1) The operator (call it Ω) puts certain numbers into the output, conditional on certain finite sets of numbers being contained in the input. (If the finite set is empty, numbers are placed unconditionally in the output.) E.g., Φ puts 1 in unconditionally, and $2m$ conditionally upon $\{m\}$. Ψ outputs m conditionally upon $\{m\}$, and outputs $m \cdot n$ conditionally upon $\{m,n\}$. Θ outputs $(0,1)$ unconditionally, and $(n, n \cdot l)$ conditionally upon $\{(n-1,l)\}$. Hence all the operators are *continuous* in the following sense:

$$n \in \Omega(x) \Leftrightarrow (\exists \text{ finite } s)[s \subseteq x \wedge n \in \Omega(s)]$$

(2) Suppose we code each "instruction"

> put m into the output conditional upon $\{a_1, \ldots, a_k\} = s$ being contained in the input

as a number, denoted

$$[\![s \to m]\!]$$

(read, *s produces m*). Then Φ, Ψ, Θ, Γ all can be coded as *recursively enumerable* sets of instructions. We define an *enumeration operator* as a continuous operator that can be so coded. In other words, if W is a recursively enumerable set, all of whose elements are of the form $[\![s \to m]\!]$, we define the associated enumeration operator by this rule (where $m \in \mathbb{N}$, $x \subseteq \mathbb{N}$, and s is a finite subset of $\mathbb{N}$):

$$m \in \Omega(x) \Leftrightarrow (\exists \text{ finite } s)[s \subseteq x \wedge [\![s \to m]\!] \in W]$$

An enumeration operator always has a unique minimal fixed point, which is always a recursively enumerable set; this is proved in Rogers, but the proof is straightforward.

5 Scott's model of the lambda calculus

In Scott's model, every closed lambda term has an interpretation, not just the normalizable ones. The trick is to use a different domain, and a different way to code procedures.

Recall the definition of a *continuous* operator $\Omega : \mathcal{P}\mathbb{N} \to \mathcal{P}\mathbb{N}$: for all $n \in \mathbb{N}$ and all $x \subseteq \mathbb{N}$,

$$m \in \Omega(x) \Leftrightarrow (\exists \text{ finite } s)[s \subseteq x \wedge m \in \Omega(s)]$$

(Immediate consequence: continuous operators are *monotone*, i.e., $x \subseteq y \Rightarrow \Omega(x) \subseteq \Omega(y)$.) So a continuous operator Ω is specified completely by the set of all "instructions"

$$\text{graph}(\Omega) = \{[\![s \to m]\!] | m \in \Omega(s), s \text{ finite} \subseteq \mathbb{N}\}$$

(Of course, this notion of "graph" differs from the one we mentioned earlier, the set of coded pairs $(n, f(n))$ for a single-valued f.)

The symbol $[\![s \to m]\!]$ stands for an encoding of the pair (s, m) as a number. To be explicit, let $\{e_k\}$ be an enumeration of all finite sets of numbers—just use binary notation, so 0 codes $\varnothing$, 1 codes $\{0\}$, 2 codes $\{1\}$, 3 codes $\{0, 1\}$, etc. Then $[\![e_k \to m]\!]$ is the pair $\langle k, m\rangle$ coded as a single number (k, m). This is a tidy coding in a couple of ways: (a) it is a 1–1 correspondence between $\mathbb{N}$ and the set of all possible instructions $[\![e_k \to m]\!]$; (b) since $k, m \leq (k, m)$, and all elements of e_k are smaller (usually *much* smaller) than k, $[\![e_k \to m]\!]$ is never an element of e_k. (This proves useful later.)

We classify the instructions in graph(Ω) into two types. If $e_k \subset e_l$ with both $[\![e_k \to m]\!]$ and $[\![e_l \to m]\!]$ belonging to graph(Ω), then we say $[\![e_l \to m]\!]$ is *superfluous*, since its presence can be inferred from the monotonicity of Ω. Non-superfluous instructions are *essential*.

Any collection x of instructions specifies a continuous operator Ω_x: just define

$$m \in \Omega_x(y) \Leftrightarrow (\exists e_k)[e_k \subseteq y \wedge [\![e_k \to m]\!] \in x]$$

The graph of Ω_x is readily obtained from x: just throw in all $[\![e_l \to m]\!]$ for which $e_k \subseteq e_l$ and $[\![e_k \to m]\!] \in x$. (I.e., all the instructions implied by instructions belonging to x.) So the graph of Ω_x is the largest possible set of instructions for Ω_x. On the other hand, if we remove all superfluous instructions from x, we get the smallest possible set of instructions. Both are convenient in that $\Omega_x = \Omega_y$ iff their largest (respectively smallest) sets of instructions are equal.

We are now nearly ready to specify Scott's model.

1. The domain $\mathcal{D}$ is $\mathcal{P}\mathbb{N}$.
2. Application xy is defined by $xy = \Omega_x(y)$.
3. Interpretaion $\mathcal{V}(\sigma)$ is defined by induction on the complexity of σ (details below).

So the key idea is, any subset of $\mathbb{N}$ can be viewed both as *data* and as *program*.

OK, let's turn to $\mathcal{V}$. Obviously we want to set $\mathcal{V}(\sigma\tau)$ equal to $\mathcal{V}(\sigma)\mathcal{V}(\tau)$, i.e., to $\Omega_{\mathcal{V}(\sigma)}\mathcal{V}(\tau)$. No problem there. What about $\lambda x.\sigma[x]$? If we've already defined $\mathcal{V}(\sigma[d])$ for every $d \subseteq \mathbb{N}$, and if $d \mapsto \mathcal{V}(\sigma[d])$ is a continuous operator Ω, then we have an obvious choice: set

$$\mathcal{V}(\lambda x.\sigma[x]) = \text{graph}(\Omega)$$

For an inductive argument, we can't confine ourselves to closed lambda terms; we have to extend $\mathcal{V}$ to lambda terms with free variables, say $\sigma[\vec{x}]$, where $\vec{x} = (x_1, \ldots, x_n)$. Note that we've already committed notational abuse above: $\mathcal{V}(\sigma[d])$ doesn't really make sense, since $\sigma[d]$ is not a closed lambda term. Really we should write $\mathcal{V}(\sigma[x])(d)$, where x is a variable (not an element of $\mathcal{D}$), $\mathcal{V}(\sigma[x])$ is an operator, and so $\mathcal{V}(\sigma[x])(d)$ is an element of $\mathcal{D}$.

We will let $\mathcal{V}(\sigma[\vec{x}])$ be an n-ary operator

$$\mathcal{V}(\sigma[\vec{x}]) : \mathcal{D}^n \to \mathcal{D}$$

So when σ is closed, we have the special case of a 0-ary operator, i.e., just an element of $\mathcal{D}$. For $\mathcal{V}(\sigma[\vec{x}])$ to be well-defined, we need to impose a standard ordering on all the variables of the lambda language; then the ordering of the arguments in $\mathcal{V}(\sigma[\vec{x}])$ will follow that ordering, regardless of how the variables appear in the form $\sigma[\vec{x}]$.

We say an n-ary operator Ω is *continuous* if the following holds: for all $m \in \mathbb{N}$, $\vec{d} \in \mathcal{D}^n$,

$$m \in \Omega(\vec{d}) \Leftrightarrow (\exists\ n\text{-tuple } \vec{s} \text{ of finite sets})[m \in \Omega(\vec{s}) \wedge \vec{s} \subseteq \vec{d}]$$

where $\vec{s} \subseteq \vec{d}$ is meant elementwise, i.e., $e_i \subseteq d_i$ for $1 \leq i \leq n$. We will also say that "$m \in \Omega(\vec{d})$" is *finitely supported*, with *support* $\vec{s}$. Note that continuous operators are monotone, in the sense that if $\vec{c} \subseteq \vec{d}$ elementwise, then $\Omega(\vec{c}) \subseteq \Omega(\vec{d})$.

An observation that will prove important very soon: say $m_1, \ldots, m_r \in \Omega(\vec{d})$ for continuous Ω. Then there is a single finite support tuple $\vec{s}$ that works simultanously for all the m_i, i.e., $m_i \in \Omega(\vec{s})$ for all m_i. This follows immediately from the monotonicity of Ω.

Now we define $\mathcal{V}(\sigma[\vec{x}])$ by induction on the complexity of $\sigma[\vec{x}]$; $\mathcal{V}(\sigma[\vec{x}])$ will be a continuous operator.

Case 0: The base case. If σ is just x, i.e., a variable, we let $\mathcal{V}(x)$ be the identity operator.

Case 1: $\mathcal{V}(\sigma[\vec{x}]\tau[\vec{y}])$. So $\mathcal{V}(\sigma[\vec{x}])$ and $\mathcal{V}(\tau[\vec{y}])$ have already been defined as continuous operators. Let $\vec{z}$ be the union of $\vec{x}$ and $\vec{y}$ (i.e., it contains all the free variables of $\sigma[\vec{x}]$ and of $\tau[\vec{y}]$, with no duplicates). Say that $\vec{z}$ has length n, and that $\vec{d} \in \mathcal{D}^n$. Let $\vec{d_x}$ correspond to the variables $\vec{x}$ and $\vec{d_y}$ to the variables $\vec{y}$. Then $\mathcal{V}(\sigma[\vec{x}])(\vec{d_x})$ and $\mathcal{V}(\tau[\vec{y}])(\vec{d_y})$ are elements of $\mathcal{D}$, and so

$$\vec{d} \mapsto \mathcal{V}(\sigma[\vec{x}])(\vec{d_x})\ \mathcal{V}(\tau[\vec{y}])(\vec{d_y})$$

is an n-ary operator; we let $\mathcal{V}(\sigma[\vec{x}]\tau[\vec{y}])$ be this operator. We need to show it's continuous.

Let $\Phi = \mathcal{V}(\sigma[\vec{x}])$, $\Psi = \mathcal{V}(\sigma[\vec{y}])$, $u = \Phi(\vec{d_x})$, and $v = \Psi(\vec{d_y})$. Suppose $m \in uv$. By the definition of application, we have $[\![e_k \to m]\!] \in u$ for some $e_k \subseteq v$. By our inductive hypothesis about the continuity of Ψ and our observation above, there is a finite support tuple $\vec{s}_y$ for all the elements of e_k, i.e., $e_k \subseteq \Psi(\vec{s}_y)$. By the continuity of Φ, $[\![e_k \to m]\!] \in \Phi(\vec{s}_x)$ for some finite support tuple $\vec{s}_x$. Using the observation again, we can meld $\vec{s}_x$ and $\vec{s}_y$ into a single finite support tuple $\vec{s}$, so $m \in \mathcal{V}(\sigma[\vec{x}]\tau[\vec{y}])(\vec{s})$. This proves the $\Rightarrow$ direction of continuity. The $\Leftarrow$ direction is similar but easier.

Case 2: $\mathcal{V}(\lambda x.\sigma[x, \vec{y}])$. Suppose $\vec{y}$ is an n-tuple. By inductive hypothesis, we have an $(n+1)$-ary continuous operator $\Phi = \mathcal{V}(\sigma[x, \vec{y}])$. Then for each $\vec{d} \in \mathcal{D}^n$, $c \mapsto \Phi(c, \vec{d})$ is a unary operator, easily seen to be continuous. Denote it by $\Phi_{\vec{d}}$. Let $\mathcal{V}(\lambda x.\sigma[x, \vec{y}])$ be the n-ary operator

$$\vec{d} \mapsto \mathrm{graph}(\Phi_{\vec{d}})$$

We just need to verify that this is continuous. Suppose $\Psi = \mathcal{V}(\lambda x.\sigma[x, \vec{y}])$ and $r \in \Psi(\vec{d})$. That is, $r \in \mathrm{graph}(\Phi_{\vec{d}})$, i.e.,

$$r = [\![e_k \to m]\!] \in \mathrm{graph}(\Phi_{\vec{d}})$$

By the definition of graph, $m \in \Phi_{\vec{d}}(e_k)$, i.e., $m \in \Phi(e_k, \vec{d})$. Since Φ is continuous, the $(n+1)$-tuple $(e_k, \vec{d})$ contains a finite $(n+1)$-tuple $(e_l, \vec{s})$ with $m \in \Phi(e_l, \vec{s})$ (so $e_l \subseteq e_k$ and $\vec{s} \subseteq \vec{d}$ elementwise). By monotonicity, $m \in \Phi(e_k, \vec{s})$, i.e., $m \in \Phi_{\vec{s}}(e_k)$. So

$$r = [\![e_k \to m]\!] \in \mathrm{graph}(\Phi_{\vec{s}})$$

i.e., $r \in \Psi(\vec{s})$ with $\vec{s} \subseteq \vec{d}$. That's what we had to prove.

Suppose Φ is a continuous unary operator, and $d \in \mathcal{P}\mathbb{N}$. Then

$$\Phi(d) = \mathrm{graph}(\Phi)d$$

This follows immediately from the definitions of continuity, graph, and application. Now suppose $\Phi = \mathcal{V}(\sigma[x])$ and $d = \mathcal{V}(\tau)$. By definition, $\mathcal{V}(\lambda x.\sigma[x]) = \mathrm{graph}(\Phi)$, so the right hand side is $\mathcal{V}(\lambda x.\sigma[x])\mathcal{V}(\tau)$, which equals $\mathcal{V}((\lambda x.\sigma[x])\tau)$. The left hand side *ought* to be equal to $\mathcal{V}(\sigma[\tau])$. If so, then the displayed equation becomes

$$\mathcal{V}(\sigma[\tau]) = \mathcal{V}((\lambda x.\sigma[x])\tau)$$

i.e., $\mathcal{V}$ respects the conversion rule of the lambda calculus.

Focussing on the left hand side, we have two kinds of substitution, or "plugging in":

- Plugging in a closed term τ into a term $\sigma[x]$ with a free variable, to get a closed term $\sigma[\tau]$.
- Plugging in a value $d = \mathcal{V}(\tau)$ for the input to an operator $\Phi = \mathcal{V}(\sigma[x])$, to get a result $\Phi(d)$.

Both are defined by induction on complexity, in pretty much the same way, so we expect they will always give the same result. If you insist on seeing the details, they go like this.

First, we have to generalize: if $\sigma[x, \vec{y}]$ is a lambda term with $n+1$ free variables $(x, \vec{y})$, and τ is a closed lambda term, then for any $\vec{c} \in \mathcal{D}^n$, the following holds:

$$\mathcal{V}(\sigma[\tau, \vec{y}])(\vec{c}) = \mathcal{V}(\sigma[x, \vec{y}])(\mathcal{V}(\tau), \vec{c})$$

Introducing the notation $\Phi = \mathcal{V}(\sigma[x, \vec{y}])$, $\Phi_\tau = \mathcal{V}(\sigma[\tau, \vec{y}])$, $d = \mathcal{V}(\tau)$, we can rewrite this as

$$\Phi_\tau(\vec{c}) = \Phi(d, \vec{c})$$

for all $\vec{c} \in \mathcal{D}^n$.

Proof is by induction of the complexity of σ. If σ is a variable, then $\vec{y}$ is empty. Also $\sigma[\tau] = \tau$ and $\Phi_\tau = \mathcal{V}(\tau) = d$; also Φ is the identity and so $\Phi(d) = d$.

If $\sigma[x, \vec{y}]$ is $(\sigma_1[x, \vec{y}]\sigma_2[x, \vec{y}])$, then for all $\vec{c} \in \mathcal{D}^n$, $\Phi_\tau(\vec{c}) = \Phi_{\tau,1}(\vec{c})\Phi_{\tau,2}(\vec{c})$, where of course $\Phi_{\tau,1}$ and $\Phi_{\tau,2}$ are $\mathcal{V}(\sigma_1[\tau, \vec{y}])$ and $\mathcal{V}(\sigma_2[\tau, \vec{y}])$. By induction, $\Phi_{\tau,1}(\vec{c}) = \Phi_1(d, \vec{c})$ and likewise for Φ_2, where naturally $\Phi_1 = \mathcal{V}(\sigma_1[x, \vec{y}])$ and $\Phi_2 = \mathcal{V}(\sigma_2[x, \vec{y}])$. But

$$\mathcal{V}(\sigma[x, \vec{y}]) \equiv \mathcal{V}(\sigma_1[x, \vec{y}])\mathcal{V}(\sigma_2[x, \vec{y}])$$

by the inductive definition of $\mathcal{V}$, so the two sides are equal in this case.

Finally, say $\sigma[x, \vec{y}]$ is $\lambda y_0.\omega[x, y_0, \vec{y}]$, where y_0 is a variable different from x and all the y's in $\vec{y}$. Let $\Psi_\tau = \mathcal{V}(\omega[\tau, y_0, \vec{y}])$ and let $\Psi = \mathcal{V}(\omega[x, y_0, \vec{y}])$. By inductive assumption,

$$\Psi_\tau(c_0, \vec{c}) = \Psi(d, c_0, \vec{c})$$

for any $c_0 \in \mathcal{D}$, $\vec{c} \in \mathcal{D}^n$, and $d = \mathcal{V}(\tau)$. But then

$$\text{graph}[c_0 \mapsto \Psi_\tau(c_0, \vec{c})] = \text{graph}[c_0 \mapsto \Psi(d, c_0, \vec{c})]$$

for any $\vec{c} \in \mathcal{D}^n$ and $d = \mathcal{V}(\tau)$. So by definition of $\mathcal{V}$ for $\lambda y_0 \ldots$,

$$\mathcal{V}(\lambda y_0.\omega[\tau, y_0, \vec{y}])(\vec{c}) = \mathcal{V}(\lambda y_0.\omega[x, y_0, \vec{y}])(d, \vec{c})$$

for $d = \mathcal{V}(\tau)$ and any $\vec{c} \in \mathcal{D}^n$. The induction is complete.

One further generalization fully justifies lambda conversion: allow τ to a be a lambda term with free variables. The "plugging in" claim becomes

$$\mathcal{V}(\sigma[\tau[\vec{z}], \vec{y}])(\vec{a}, \vec{c}) = \mathcal{V}(\sigma[x, \vec{y}])(d, \mathcal{V}(\tau[\vec{z}])(\vec{a}), \vec{c})$$

The proof is essentially the same, except more complicated.

6 Partial functions

Partial functions could be coded in the Scott model as single-valued sets, but there is another approach that works better.

Let $p : \mathbb{N} \to \mathcal{P}\mathbb{N}$; this includes partial functions, which are just those functions from $\mathbb{N}$ to $\mathcal{P}\mathbb{N}$ for which $|p(m)| \leq 1$ for all $m \in \mathbb{N}$. Define $\hat{p} : \mathcal{P}\mathbb{N} \to \mathcal{P}\mathbb{N}$ by

$$\hat{p}(x) = \bigcup_{i \in x} p(i)$$

The operator $\hat{p}$ is *distributive*, which means that:

$$\hat{p}(\varnothing) = \varnothing$$

$$\hat{p}\left(\bigcup_{\alpha} d_\alpha\right) = \bigcup_{\alpha} \hat{p}(d_\alpha)$$

Every distributive operator is $\hat{p}$ for some $p : \mathbb{N} \to \mathcal{P}\mathbb{N}$. Distributive implies continuous. The graph of a distributive operator is distinguished by the fact that all essential instructions are of the form $[\![\{i\} \to m]\!]$, $i, m \in \mathbb{N}$. (Hence there are no unconditional instructions, i.e., those of the form $[\![\varnothing \to m]\!]$.) If we had an essential instruction of the form $[\![\{i, j\} \to m]\!]$, that would mean that i and j "cooperated" to produce m. Continuity means that there is no "infinite cooperation", and distributivity means that there is no cooperation at all (and no unconditional instructions).

7 Examples

From now on, I won't bother writing $\mathcal{V}$. If I write (say) $\lambda x.xx$, this can stand either for the lambda term or its interpretation, depending on context. (Usually I'll mean the interpretation.)

We examine the machinery of Scott's model in detail for some examples involving $I = \lambda x.x$ and $D = \lambda x.xx$.

First let's figure out what the graph of I looks like.

$$[\![e_k \to m]\!] \in \text{graph}(I) \Leftrightarrow m \in I(e_k) = e_k$$

So if $e_k = \{m_1, \ldots, m_r\}$, then $[\![e_k \to m_i]\!]$ is in graph(I) for each i. From this it's easy to see that the essential instructions are just all those of the form

$$[\![\{m\} \to m]\!]$$

In other words, the instructions tell I to pass along any element of the input set directly to the output set, without modification. Since I itself is a subset of $\mathbb{N}$, when in the "data" position, the equation $II = I$ should cause no discombobulation.

Next, let's figure out the graph of D.

$$[\![e_n \to m]\!] \in \text{graph}(D) \Leftrightarrow m \in e_n e_n \Leftrightarrow (\exists e_k)\,([\![e_k \to m]\!] \in e_n \wedge e_k \subseteq e_n)$$

So we find essential instructions with the following steps:

1. Start by picking a finite set e_k.
2. Pick a number m.
3. Form $r = [\![e_k \to m]\!]$.
4. Let $e_n = \{r\} \cup e_k = \{[\![e_k \to m]\!]\} \cup e_k$.

5. The essential instruction is $[\![e_n \to m]\!]$.

So

$$[\![e_k \cup \{[\![e_k \to m]\!]\} \to m]\!]$$

is our instruction, which we may read

> If $e_k \subseteq x$ and x contains the instruction $[\![e_k \to m]\!]$, then put m into the output.

By an earlier observation about the coding $[\![e_k \to m]\!]$, we see that this instruction will never be an element of e_k. (As we noted, the pair (k, m) is $\geq k, m$, and all elements of e_k are smaller than k.)

Now let's work out $DI = II = I$. First we show that $DI \supseteq I$. D searches I looking for e_k and m such that $e_k \subseteq I$ and $[\![e_k \to m]\!] \in I$. On success, D places m into DI. The essential instructions of I are of the form $[\![\{i\} \to i]\!]$. So if we let $e_k = \{i\}$ with $i \in I$, then we have success and i is placed in the output. That is, all elements of I are placed in the output, so $DI \supseteq I$.

Next: can a superfluous instruction of I add anything else to the output DI? Say $[\![e_k \to m]\!] \in I$. As we saw, that means that $m \in e_k$, so if we also have $e_k \subseteq I$, then $m \in I$. So $DI \subseteq I$.

Now we turn to DD. D searches D looking for instructions $[\![e_n \to m]\!] \in D$ with $e_n \subseteq D$. Recall that

$$[\![e_n \to m]\!] \in D \Leftrightarrow (\exists e_k)\,([\![e_k \to m]\!] \in e_n \wedge e_k \subseteq e_n)$$

So if such an $[\![e_n \to m]\!]$ exists, then also there is an $e_k \subseteq e_n \subseteq D$ and $[\![e_k \to m]\!] \in e_n \subseteq D$. Thus $[\![e_k \to m]\!] \in D$ and $e_k \subseteq D$. By the remark on coding, $k < n$. In other words, if we have an e_n satisfying the requirements, we have an e_k with $k < n$ also satisfying the requirements. Conclusion: there is no such e_n. So D never succeeds in its search, and $DD = \varnothing$.

Contrast with the index model, where DD is unnormalizable and thus has no interpretation.

8 Fixed points

Now we look at

$$[\lambda x.u(xx)][\lambda x.u(xx)]$$

the so-called fixed-point combinator. Writing δ for $\lambda x.u(xx)$, we have:

$$\begin{aligned}\delta\delta &= [\lambda x.u(xx)][\lambda x.u(xx)] \\ &= u\,([\lambda x.u(xx)][\lambda x.u(xx)]) \\ &= u(\delta\delta)\end{aligned}$$

so $\delta\delta$ is a fixed point for u—in any model of the lambda calculus in which it is interpreted.

To illustrate, consider the index model. Then δ is an index for the partial function $x \mapsto \varphi_u(\varphi_x(x))$, so the program P_δ has instructions to generate the program P_x for φ_x, then call P_x with input x, and pass the result to P_u (the program for φ_u). That is, φ_δ sets up the program $P_u(P_x(x))$ and then starts executing it. Hence $\varphi_\delta(\delta)$ first sets up $P_u(P_\delta(\delta))$ and then starts to execute it; the first stage of execution is to compute $\varphi_\delta(\delta)$, so $P_\delta(\delta)$ loops, $\varphi_\delta(\delta)$ diverges, and $\delta\delta$ is not interpreted in the index model.

However, if we follow the divergence of $\varphi_\delta(\delta)$, we find that these programs are generated at these stages:

Stage 1:	$P_u(P_\delta(\delta))$
Stage 2:	$P_u(P_u(P_\delta(\delta)))$
Stage 3:	$P_u(P_u(P_u(P_\delta(\delta))))$
etc.	

Suppose at stage k the execution of the innermost $P_\delta(\delta)$ is artificially interrupted and 0 arbitrarily chosen as the output. Then the final output would be $\varphi_u^k(0)$. This "approximate fixed point" might sometimes be good enough.

We illustrate with the Pow2 example, with the enumeration operator Φ, from p.8. Pow2 $= W_t = \mathrm{dom}(\varphi_t)$ for some t. W_t will be a fixed point for φ_u (i.e., $W_t = W_{\varphi_u(t)}$) if φ_u is constructed to represent Φ in the sense that for all $n, x \in \mathbb{N}$,

$$n \in W_{\varphi_u(x)} \Leftrightarrow (n = 1 \vee (n \in \mathrm{Even} \wedge (n/2) \in W_x))$$

Now, φ_u is just a syntactic transformation of programs, and amounts to adding a prefix to P_x (regarding P_x as an "acceptor" for W_x). If $n \leq 2^{k-1}$, then $\varphi_u^k(0)$ correctly decides if $n \in$ Pow2, i.e.,

$$n \leq 2^{k-1} \Rightarrow \left(n \in \mathrm{Pow2} \Leftrightarrow n \in W_{\varphi_u^k(0)}\right)$$

In Scott's model, $[\lambda x.u(xx)][\lambda x.u(xx)]$ is the minimal fixed point of u:

$$\mathrm{fix}(u) = \bigcup_{k \in \mathbb{N}} u^k \varnothing$$

To smooth away technicalities, we make some definitions first. For all $u \in \mathcal{P}\mathbb{N}$, $\bar{u}$ is the "standard form" for u, obtained by adding in all the superfluous instructions. (So $\bar{u} = \mathrm{graph}(\lambda x.ux)$.)

Define

$$[\![e_k \to e_m]\!] = \{[\![e_k \to h]\!] | h \in e_m\}$$

so both arguments and result are finite subsets of $\mathbb{N}$. E.g., $[\![e_k \to \varnothing]\!] = \varnothing$. (Note: $[\![\varnothing \to e_m]\!]$ won't always be empty.) Note that this opens to door to, e.g., $[\![\, [\![e_i \to e_j]\!] \to e_k]\!]$. We have

$$e_m \subseteq xy \Leftrightarrow (\exists e_n)\,([\![e_n \to e_m]\!] \subseteq \bar{x} \wedge e_n \subseteq y)$$

(e.g., $e_m \subseteq [\![e_n \to e_m]\!] e_n$; in fact, $e_m = [\![e_n \to e_m]\!] e_n$)

$$[\![e_n \to e_m]\!] \subseteq \lambda x.[\ldots x \ldots] \Leftrightarrow e_m \subseteq [\ldots e_n \ldots]$$

Let $\delta = \lambda x.u(xx)$. Below, y is any subset of $\mathbb{N}$.

$$\begin{aligned}[\![e_n \to e_m]\!] \subseteq \delta &\Leftrightarrow e_m \subseteq u(e_n e_n) \\ &\Leftrightarrow (\exists e_k)\,([\![e_k \to e_m]\!] \subseteq \bar{u} \wedge e_k \subseteq e_n e_n) \\ &\Leftrightarrow (\exists e_k, e_l)\,([\![e_k \to e_m]\!] \subseteq \bar{u} \wedge [\![e_l \to e_k]\!] \subseteq \overline{e_n} \wedge e_l \subseteq e_n)\end{aligned}$$

Also,

$$\begin{aligned}e_m \subseteq \delta y &\Leftrightarrow (\exists e_n)\,([\![e_n \to e_m]\!] \subseteq \delta \wedge e_n \subseteq y) \\ &\Leftrightarrow (\exists e_n, e_k, e_l)\,([\![e_k \to e_m]\!] \subseteq \bar{u} \wedge [\![e_l \to e_k]\!] \subseteq \overline{e_n} \wedge e_l \subseteq e_n \subseteq y) \\ &\Leftrightarrow (\exists e_k, e_l)\,([\![e_k \to e_m]\!] \subseteq \bar{u} \wedge [\![e_l \to e_k]\!] \subseteq \bar{y} \wedge e_l \subseteq y)\end{aligned}$$

(e_n is just a large enough finite piece of y).

Also,

$$e_m \subseteq \delta\delta \Leftrightarrow (\exists e_k, e_l)\,([\![e_k \to e_m]\!] \subseteq \bar{u} \wedge [\![e_l \to e_k]\!] \cup e_l \subseteq \delta)$$

(Note: $\delta = \bar{\delta}$.)

Recap:

$$\begin{aligned}&\text{(I) } [\![e_n \to e_m]\!] \subseteq \delta \Leftrightarrow (\exists e_k, e_l)\,([\![e_k \to e_m]\!] \subseteq \bar{u} \wedge [\![e_l \to e_k]\!] \subseteq \overline{e_n} \wedge e_l \subseteq e_n) \\ &\text{(II) } e_m \subseteq \delta\delta \Leftrightarrow (\exists e_k, e_l)\,([\![e_k \to e_m]\!] \subseteq \bar{u} \wedge [\![e_l \to e_k]\!] \cup e_l \subseteq \delta)\end{aligned}$$

To generate a typical group of instructions in δ, we start with e_k, e_m such that $[\![e_k \to e_m]\!] \subseteq \bar{u}$. We then pick e_l arbitrarily and have $[\![\, [\![e_l \to e_k]\!] \cup e_l \to e_m]\!] \subseteq \delta$. (Here, $e_n = [\![e_l \to e_k]\!] \cup e_l$.) Then δ incorporates (so to speak) one application of modus ponens plus one application of u. To illustrate,

suppose u contains the unconditional instructions $[\![\varnothing \to e_m]\!]$—so $e_k = \varnothing$. Set $e_l = \varnothing$, to get $[\![\varnothing \to e_m]\!] \subseteq \delta$.

We show next that $\mathrm{fix}(u) \subseteq \delta\delta$ by a direct, grubby proof. (This will help motivate the proof that $\delta\delta \subseteq \mathrm{fix}(u)$.) If $m \in \mathrm{fix}(u)$ then there are finite sets $f_1, \ldots f_r$ such that $m \in f_r$, and $f_{i+1} \subseteq uf_i$ for all i, and $f_1 \subseteq u\varnothing$. If $f_{i+1} \subseteq uf_i$ then $[\![f_i \to f_{i+1}]\!] \subseteq \bar{u}$. We will use this to form instructions contained in δ that yield $f_1 \subseteq \delta\delta$, $f_2 \subseteq \delta\delta$,…,$f_r \subseteq \delta\delta$. Define inductively $\varnothing = a_0 \subseteq a_1 \subseteq \cdots \subseteq a_{r-1}$, all finite sets, so that for all i, $[\![a_i \to f_{i+1}]\!] \subseteq \delta$ and $a_i \subseteq \delta$. In fact, set

$$a_{i+1} = a_i \cup [\![a_i \to f_{i+1}]\!]$$

Since $a_i = a_{i-1} \cup [\![a_{i-1} \to f_i]\!]$,

$$[\![a_i \to f_{i+1}]\!] = [\![a_{i-1} \cup [\![a_{i-1} \to f_i]\!] \to f_{i+1}]\!]$$

which, since $[\![f_i \to f_{i+1}]\!] \subseteq \bar{u}$, fits the paradigm of "one application of modus ponens plus one application of u", and so is a subset of δ. So if $a_i \subseteq \delta$ then $a_i \cup [\![a_i \to f_{i+1}]\!] \subseteq \delta$. We therefore have: $a_i \subseteq a_{i+1}$, $[\![a_i \to f_{i+1}]\!] \subseteq \delta$, and $a_i \subseteq \delta$. Therefore $f_{i+1} \subseteq \delta\delta$.

E.g.,

$$\begin{aligned} a_1 &= [\![\varnothing \to f_1]\!] \\ a_2 &= [\![\, [\![\varnothing \to f_1]\!] \to f_2]\!] \cup a_1 \\ a_3 &= [\![\, [\![\varnothing \to f_1]\!] \cup [\![\, [\![\varnothing \to f_1]\!] \to f_2]\!] \to f_3]\!] \cup a_2 \\ &\text{etc.} \end{aligned}$$

Next: $\delta\delta \subseteq \mathrm{fix}(u)$. Let f be a finite set. We wish to induct on something to show that $f \subseteq \delta\delta \Rightarrow f \subseteq \mathrm{fix}(u)$. Obvious choice for induction: the $[\![a_i \to f_{i+1}]\!]$ sets, which become more complicated as i increases.

Lemma: For all $h \in \mathbb{N}$, if $[\![e_\ell \to h]\!] \in e_i$, then $\ell < i$.

Proof: $[\![e_\ell \to h]\!]$ is really the coded pair (ℓ, h), and $(\ell, h) \in e_i$, so $\ell \leq (\ell, h) < i$.

Lemma: For all finite $f \subseteq \mathbb{N}$, if $e_i \cup [\![e_i \to f]\!] \subseteq \delta$, then $f \subseteq \mathrm{fix}(u)$.

Proof: Induction on i. We have

$$[\![e_i \to f]\!] \subseteq \delta \Rightarrow (\exists e_k, e_l)\,([\![e_k \to f]\!] \subseteq \bar{u} \wedge e_l \subseteq e_i \wedge [\![e_l \to e_k]\!] \subseteq \overline{e_i})$$

Now $e_i \subseteq \delta$, so

$$[\![e_k \to f]\!] \subseteq \bar{u} \wedge e_l \subseteq e_i \subseteq \delta \wedge [\![e_l \to e_k]\!] \subseteq \overline{e_i} \subseteq \delta$$

(The idea is to pursue this analogy: $f \sim f_{i+1}$, $e_i \sim a_i$, $e_k \sim f_i$, $e_l \sim a_{i-1}$. We don't have $e_i = e_l \cup [\![e_l \to e_k]\!]$, but only $e_l \subseteq e_i$ and $[\![e_l \to e_k]\!] \subseteq \overline{e_i}$.)

If $e_k = \varnothing$ then $f \subseteq u\varnothing$. Let $h \in e_k$. Since $[\![e_l \to h]\!] \subseteq \overline{e_i}$, we have $(\exists e_\ell \subseteq e_l)$ with $[\![e_\ell \to h]\!] \in e_i$. Therefore

$$\ell < i \wedge e_\ell \subseteq e_l \subseteq e_i \subseteq \delta \wedge [\![e_\ell \to h]\!] \in e_i \subseteq \delta$$

so by hypothesis, $h \in \mathrm{fix}(u)$. Since h was arbitrary, $e_k \subseteq \mathrm{fix}(u)$. As $[\![e_k \to f]\!] \subseteq \bar{u}$, $f \subseteq \mathrm{fix}(u)$. QED

Theorem: If $f \subseteq \delta\delta$ then $f \subseteq \mathrm{fix}(u)$.

Proof: If $f \subseteq \delta\delta$ then $(\exists e_k, e_l)\,([\![e_k \to f]\!] \subseteq \bar{u} \wedge [\![e_l \to e_k]\!] \cup e_l \subseteq \delta)$. But if $[\![e_l \to e_k]\!] \cup e_l \subseteq \delta$ then $e_k \subseteq \mathrm{fix}(u)$, so if $[\![e_k \to f]\!] \subseteq \bar{u}$ then $f \subseteq \mathrm{fix}(u)$. QED

Shorn of all details, what is the difference between the two models of the lambda calculus? We consider the example Pow2.

In the index model: Let $\mathrm{Pow2} = W_t = \mathrm{dom}(\varphi_t)$, where t is a fixed point for φ_u, i.e., for all n, $\varphi_{\varphi_u(t)}(n) = \varphi_t(n)$. So φ_t is an "acceptor" for Pow2. As previously noted, $\varphi_u(t)$ adds a "prefix" to φ_t, which checks to see if $n = 1$, in which case the input is accepted, or if n is even, in which case $n/2$ is

passed to φ_t. Now $\varphi_\delta(\delta)$ diverges, and so cannot give us t. If we try to compute $\varphi_{\varphi_\delta(\delta)}(n)$, then we must finish computing $\varphi_\delta(\delta)$ (which we cannot) before we start looking at n, even though for any n in Pow2, a finite part of the computation for $\varphi_\delta(\delta)$ would be enough to tell $\varphi_{\varphi_\delta(\delta)}(n)$ to accept n.

In Scott's model: Pow2 $= \delta\delta$, where $u = \text{graph}(\Phi)$ (with the Φ from p.8). Acceptance here is $y \in \delta\delta$. In $\delta\delta$, so to speak, all the computations are done *simultaneously*. We should think of Scott's model as dynamically growing, only a finite amount being present at any one time. Once a finite part of δ is computed, however, we can use it right away—we don't have to wait until δ is complete to start forming $\delta\delta$. Thus, in the index model, as the computation $\varphi_\delta(\delta)$ proceeds, we learn more and more of what $\varphi_{\varphi_\delta(\delta)}$ "would look like" if $\varphi_\delta(\delta)$ ever converged. Since it doesn't, this knowledge is useless. So $\delta\delta$ is not a fixed point in the index model. Scott's model avoids this impasse.

One last example of a fixed point. Recall that if $p : \mathbb{N} \to \mathcal{P}\mathbb{N}$, we define $\hat{p} : \mathcal{P}\mathbb{N} \to \mathcal{P}\mathbb{N}$ by

$$\hat{p}(x) = \bigcup_{i \in x} p(i)$$

Suppose $u \in \mathcal{P}\mathbb{N}$. Then u defines a function $p_u : \mathbb{N} \to \mathcal{P}\mathbb{N}$ by $p_u(n) = u\{n\}$, for all $n \in \mathbb{N}$. (We'll just write un instead of $u\{n\}$, and similarly for other singletons.) Let $\$u = \text{graph}(\widehat{p_u})$. We also use the suggestive notation

$$\$u = \lambda n \in \mathbb{N}.un$$

So $\$u$ agrees with u when the arguments are numbers, and $\$u$ is the graph of a distributive function

$$(\$u)x = \bigcup_{i \in x} ui$$

The operator \$ may be regarded as the limit of a sequence of operators s_n, where

$$(s_n u)x = \bigcup_{i \in x \wedge i < n} ui$$

(Scott's notation for $s_n u$ is $\langle u_0, \ldots, u_{n-1} \rangle$.)

We can define s_n in terms of s_{n-1} with the aid of a few auxiliary concepts. For any $t \subseteq \mathbb{N}$, let $t+1 = \{i+1 | i \in t\}$. Given $u \in \mathcal{P}\mathbb{N}$, let $v = \lambda t.u(t+1)$; then for all i, $vi = u(i+1)$. For any $x \in \mathcal{P}\mathbb{N}$, set $x - 1 = \{i | i+1 \in x\}$. Then

$$\begin{aligned}(s_n u)x &= (\text{if } 0 \in x \text{ then } u0 \text{ else } \varnothing) \cup \bigcup_{i \in x \wedge 0 < i < n} ui \\ &= (\text{if } 0 \in x \text{ then } u0 \text{ else } \varnothing) \cup \bigcup_{i+1 \in x \wedge i < n-1} u(i+1) \\ &= (\text{if } 0 \in x \text{ then } u0 \text{ else } \varnothing) \cup \bigcup_{i \in x-1 \wedge i < n-1} vi \\ \text{where } v &= \lambda t.u(t+1)\end{aligned}$$

so

$$\begin{aligned}(s_n u)x &= (\text{if } 0 \in x \text{ then } u0 \text{ else } \varnothing) \cup (s_{n-1} v)(x-1) \\ &= (\text{if } 0 \in x \text{ then } u0 \text{ else } \varnothing) \cup (s_{n-1}(\lambda t.u(t+1)))(x-1)\end{aligned}$$

This combination of "if-then-else" with a union, where you put something in the output if $0 \in x$ and other things as well if x contains positive numbers, occurs often enough to inspire a notation:

$$(x \supset y, z) = (\text{if } 0 \in x \text{ then } y \text{ else } \varnothing) \cup (\text{if } x - 1 \neq \varnothing \text{ then } z \text{ else } \varnothing)$$

So

$$\begin{aligned}(s_n u)x &= x \supset u0, (s_{n-1}(\lambda t.u(t+1)))(x-1) \\ s_n &= \lambda u \lambda x[x \supset u0, (s_{n-1}(\lambda t.u(t+1)))(x-1)]\end{aligned}$$

Thus we can introduce something that summarizes the transition from s_{n-1} to s_n:

$$\Sigma s = \lambda u \lambda x[x \supset u0, (s(\lambda t.u(t+1)))(x-1)]$$

Now, one can show that \$ is the unique minimal fixed point of Σ. That it *is* a fixed point is fairly clear:

$$\begin{aligned}(\$u)x &= \bigcup_{i\in x} ui \\ &= (\text{if } 0 \in x \text{ then } u0 \text{ else } \varnothing) \cup \bigcup_{i\in x\wedge 0<i} ui \\ &= (\text{if } 0 \in x \text{ then } u0 \text{ else } \varnothing) \cup \bigcup_{i\in x-1} u(i+1) \\ &= x \supset u0, (\$(\lambda t.u(t+1)))(x-1) \\ &= ((\Sigma\$)u)x\end{aligned}$$

That it is the unique minimal fixed point is more tedious. Still, if Y is the fixed point operator introduced earlier,

$$\begin{aligned}Yu &= [\lambda x.u(xx)][\lambda x.u(xx)] \\ Y &= \lambda u.[\lambda x.u(xx)][\lambda x.u(xx)]\end{aligned}$$

then

$$\$ = Y\Sigma = Y\,(\lambda s\lambda u\lambda x[x \supset u0, (s(\lambda t.u(t+1)))(x-1)])$$

We can now define $\lambda n \in \mathbb{N}.[\ldots n \ldots]$, which will always be the graph of a distributive function, by

$$\lambda n \in \mathbb{N}[\ldots n \ldots] = \$(\lambda x[\ldots x \ldots])$$

9 Submodels

The nice thing about Scott's model is that every closed lambda term is interpreted, normalizable or not. However, the model is bigger than the index model: there are $2^{\aleph_0}$ continuous operators. Can we find a countable model that still enjoys the pleasant features of Scott's model? Yes we can!

Define $\mathcal{A} \subseteq \mathcal{P}\mathbb{N}$ to be a subalgebra iff $\mathcal{A}$ is closed under application. We can define the subalgebra ςA generated by $A \subseteq \mathcal{P}\mathbb{N}$ in the usual way. (Caution: application is not associative. For example, if $G \in \mathcal{P}\mathbb{N}$, then $\varsigma\{G\}$ is not just $\{G^n | n \in \mathbb{N}\}$, since (for example) $G(GG)$ and $(GG)G$ are usually distinct.)

$\mathcal{A}$ is *closed under abstraction* iff given any closed lambda term containing only constants for elements of $\mathcal{A}$ (or else constant-free), the value of the term is in $\mathcal{A}$.

By a classic result in combinatory logic, closure under abstraction is a very mild requirement to impose on a subalgebra. Namely, define

$$\begin{aligned} S &= \lambda x \lambda y \lambda z.(xz)(yz) \\ K &= \lambda x \lambda y.x \\ I &= \lambda x.x \end{aligned}$$

Then $\mathcal{A}$ is closed under abstraction iff $S, K, I \in \mathcal{A}$. (In fact, $I = (SK)K$, so S and K are enough.)

Let RE be the class of recursively enumable sets. So RE $\subseteq \mathcal{P}\mathbb{N}$. RE is a subalgebra, and S and K belong to RE. Also $\mathbb{N} \subset$ RE, provided we elide the difference between n and $\{n\}$. RE contains combinators for three useful functions:

$$\begin{aligned} \text{suc} &= \lambda x.x + 1 \\ \text{pred} &= \lambda x.x - 1 \\ \text{cond} &= \lambda x \lambda y \lambda z.x \supset y, z \end{aligned}$$

So RE is a subalgebra that contains provisions for arithmetic constructs and branching.

Note that RE is a countable model of the lambda calculus in which every closed term has a value. Basically, RE is the collection of all enumeration operators.

In fact, RE can be characterized as the subalgebra generated by {0, suc, pred, cond, K, S}. This leads to various interesting equivalent definitions of RE and related notions. We simply state results.

1. RE is generated by a single element: $\mathrm{RE} = \varsigma\{G\}$.
2. RE is the set generated by $\{0, \mathrm{suc}, \mathrm{pred}, \mathrm{cond}\}$ under the operations of application and abstraction. Thus, let LAMBDA be the lambda language with constants 0, suc, pred, cond. RE is just the set of values of closed terms of LAMBDA. So we have a characterization of the concept "computable element of $\mathcal{P}\mathbb{N}$" that is "intrinsic to the model $\mathcal{P}\mathbb{N}$".
3. In addition to characterizing the computable *elements* of $\mathcal{P}\mathbb{N}$, we have a characterization of the computable *functions* $(\mathcal{P}\mathbb{N})^n \to \mathcal{P}\mathbb{N}$. Indeed, these are exactly the values of lambda terms with free variables, as we saw earlier.

Scott calls $\mathcal{A}$ a subalgebra iff $\mathcal{A}$ is closed under abstraction and contains the element G from item (1), so $\mathrm{RE} \subseteq \mathcal{A}$ always.

Standard recursion theory can be carried out inside RE. Thus we have an analog to the usual Gödel numbering $\{\varphi_k\}$ of partial recursive functions, call it $\mathrm{val} : \mathbb{N} \to \mathrm{RE}$. The function val has various nice features that make it easier to work with than $\{\varphi_k\}$ (or so Scott claims), e.g., $\mathrm{val} \in \mathrm{RE}$, and there is a simply constructed combinator "apply" such that

$$\mathrm{val}(\mathrm{apply}(m)(n)) = \mathrm{val}(m)\mathrm{val}(n)$$

for all numbers m and n. It is clear, however, that val is not any more "powerful" than $\{\varphi_k\}$, and anything you can do with one you can do with the other.