Computability for the Mathematical

Peter Hoffman

This is mostly a leisurely treatment of a famous discovery : the "miracle" (as Gödel termed it) that the idea of mechanical process *has an* absolute meaning*, a meaning independent of any other empirical law, or any system of logic.*

One main purpose here is to give mathematical 'completeness' to the material near the end of the elementary logic text **[LM]***, the companion work to this one. That material relates to the* incompletability of formal mathematics *discovered by Gödel. The theory of* computable functions and algorithms *is what we are referring to in the paragraph above. Pedagogically it seems to be a good idea to take some time over this theory, rather than rushing into ad hoc definitions to minimize the amount of trouble proving the basic results needed. Below we discuss the history to a tiny extent, give at least a few details on two of the original approaches from the 1930's (and later, a lot of detail on the λ-calculus approach), and then adopt a kind of 'Pascal-type software' definition. The existence of such a definition is certainly well-known to the experts. However, no logic text I know of takes this approach. There are clearly disadvantages as well as advantages. But the fact that many students have done at least some programming in high-level command languages seems to make this an advantageous framework when teaching logic.*

To summarize quickly, a command language **ATEN** *is inductively defined, whose atoms are assignment commands, and whose only command constructions are concatenation and 'while-do' commands. The semantics is very easy to define rigorously. Then one finds that* **ATEN** *computes precisely the (partial) recursive functions. It is edifying that the primitive recursive functions are computed by a sublanguage* **PTEN** *quickly described by restricting the 'while-do' commands in a rather simple manner. In fact we start with a slightly richer (syntactically only!) language* **BTEN***, which includes 'if-then-else' commands, as well as a command construction which gives some of the flexibility of variable declarations. This is better for writing the needed algorithms in the proofs of the basic theorems of Gödel, Church, and Tarski. But* **ATEN** *is of course just as strong semantically, and preferable in theoretical studies. My limited knowledge of history led me to think it appropriate to call this approach by the name "Babbage computability" (as opposed to "Turing computability", etc.).*

Another motivation for this work is to produce a somewhat detailed account, tailored to the teaching of mathematics specialists, of this major advance in our understanding of the world. There are now several very nice texts for computer science students, for example, **[DSW], [F], [J]** *and* **[Si]***. In certain respects, these books are unsuitable for students not specializing in CS, despite their many virtues.*

I owe a debt of gratitude to Phil Scott at University of Ottawa. Reacting to my query and an early version of this paper, amongst other helpful comments, he was able to point out the CS texts [**J**]**,** [**KMA**] *and* [**Som**]. *They use computer languages evidently equivalent to the formal language* **ATEN**. *To the extent given, details of their "Turing equivalence" are very different, these books being designed for CS students, as opposed to mathematics students.*

This work is written in a discursive style, at a level suitable for anyone who worked their way through [**LM**]. *In particular, the more sophisticated specialists in pure mathematics will probably find it much too wordy, though they might discover one or two things of interest not dealt with in the more advanced treatises like* [**S**] *and* [**M**].

The following quotation from [**DSW**], *referring to 'proofs' of the equivalence of various definitions of* computability, *is advice which I hope to be seen to have followed :* "...it should not be the reader's responsibility ...to fill in vaguely sketched arguments ...it is our responsibility as authors to arrange matters so that the simulations can be exhibited simply, clearly and completely."

As mentioned above, in Section VII we give a fairly detailed discussion of the λ-calculus approach to computability. Then we proceed to material on λ-models. This includes a treatment of Dana Scott's original approach, as well as some very nice theorems of Park and Wadsworth. The latter have not appeared with details elsewhere than in the original research articles, and should appeal quite strongly to most mathematicians.

Section VIII is also a quite lengthy self-contained discussion, of the use of formal logic methods in verifying that a program 'does what it's supposed to do'. The buzzwords here are Floyd-Hoare logic *and* dynamic logic. *Completely thorough treatments of soundness and relative completeness are given (apparently for the first time) for two proof systems. One is the original work where Stephen Cook discussed completeness for a language containing variable and procedure declarations with parameters. The other is a reduction to a deterministic language of an early part of Harel's thesis work on dynamic logic. Each of these needed one crucial correction which came to light in the careful checking needed to verify soundness and completeness.*

Table of Contents

We are interested in describing everything that any conceivable computer could possibly do.

Interpreted literally, this sounds overly ambitious, to say the least! Listing all possible practical applications is not what is intended. We are thinking about the 'innards' of a computing machine, which you can imagine as a bunch of boxes (storage locations), each containing a natural number (or just a 0 or a 1 if preferred). What we want to study are all the possibilities for a 'computation' (which amounts to a finite sequence of changes, carried out by some mechanical process, to what is in these boxes). So the encoding, for a practical computer application, of the actual input into a configuration of box entries (and the decoding at the end into actual output) will be largely ignored here. As we shall see, this interpretation of the question "What is computable?" may, without loss of generality, be reduced to the more specific question : "Which functions of natural numbers, returning natural numbers, are to be regarded as being computable functions?" This question cannot be answered by giving a mathematical *theorem* and its proof. It is a question of giving a mathematical *definition*—but one which has involved considerable creativity and pregnancy, not having been at all easy or obvious for the pioneers in the foundations of mathematics who stuggled with the question for decades (or perhaps centuries) before its (apparently final) resolution in the 1930's .

A related question involves the human mind (whether or not regarded as identical with the brain). Is it to count (merely) as a "conceivable computer"? This will not be dwelt on here, but certainly has elicited considerable discussion, technical literature, and writing of popular books over the years. It's the academic discipline known as thermostat.is.conscious.and.has.a.soul.ology.

The first section below contains some general discussion to organize our ideas, before producing several mathematical definitions of computability in the second section and later. Then in sections III and IV we concentrate on showing the equivalence of two of the definitions. The latter are used in V and VI to fill in the needed details in the proofs, as given in [**LM**], of Gödel's incompleteness theorem, and theorems of Church answering Hilbert's "Entscheidungsproblem", and of Tarski establishing the "undefinability of truth". Section IV also contains a treatment of *universal* programs, machines and functions, the basic theoretical model for computers, and of Turing's famous results about the halting problem (as well as an optional course in recursion theory and several related disparate topics). Finally,

VII and VIII are quite lengthy sections, being self-contained treatments of λ-calculus/models and of the logic involved in program verification.

I. Introductory remarks and informal deductions.

The following quote, from a 19th century history of biology, is a succinct description of (at least part of) one's conception of a computation being carried out:

> *. . . a device operating mechanically, that is, without occasional intervention or sustained by voluntary conscious action.*

A more detailed description (but not yet a mathematical definition) of what is involved might be the following.

What is *computable* is any 'function' for which there is an *algorithm* (or *mechanical procedure*) whose *input* the function maps to the procedure's *output*, when that algorithm/procedure is carried out by a *computing device/being*. Such a device must be, in a suitable sense, a *finite discrete object*, 'containing' the algorithm, which itself is a *finite* list of instructions. And the 'motion' within the device from input to output must take place in a *finite number of discrete steps* (but a number which, for various inputs, can vary, and might be unbounded if, as per usual, the number of possible inputs is infinite).

In this context, it is necessary to discuss the following possibilities, (i) and (ii), and their close connection to each other.

(i) Mechanisms exist which keep spitting out a piece of output every once in a while, producing an unending list of results. This idea of 'algorithmic listing' is known technically as the *recursive enumerability* or *semidecidability* or *listability* of the set of all the pieces of output. It can however be absorbed theoretically within the following second possible variation on our above rough description.

(ii) Going back to the description further up of what we shall mean by *computable*, perhaps for some input there is *no* output, in the sense that the device carrying out the algorithm would continue, without ever stopping, to take these discrete steps. Often this is conceptualized as having the algorithm be able to 'recognize' certain input, which input is not actually a member of the *domain* of the function being computed. Such a function would be called

a *partial (non-total)* function. This terminology was used on p. 403 of [**LM**] in the definition of recursive functions, whose domains are *subsets* of $\mathbf{N}^k$ for various k (a *proper* subset in the case of a non-total function), and whose values are natural numbers, i.e. in $\mathbf{N}$.

Besides the obvious picture of (i) being a procedure with natural number inputs proceeding to deal with all inputs in the usual order, here is a more interesting connection between (i) and (ii).

A mechanism as in (i) can be used to produce an algorithm as in (ii) whose computed function maps all the objects (listed by the type (i) mechanism) to a fixed answer (say, "yes"), but the function is undefined on all other objects in the larger set from which the listing occurs. The algorithm simply compares its fixed input successively to members of the list generated by the mechanism, until and if it finds an identical list member, at which point it stops and spits out the word "yes".

In the opposite direction, a mechanism for algorithmically listing the domain of a partial function, one which is computable via an algorithm as in (ii), could be conceived as follows. One assumes that the set of *all* possible inputs is algorithmically listable, and one feeds that input list into a machine in such a way that the nth piece of input begins to be processed by the given algorithm exactly after: (" $n-1$ " steps in processing the first piece of input have been completed), and (" $n-2$ " steps in processing the second piece of input have been completed), ……,and (one step in processing the $(n-1)$st piece of input have been completed). Whenever one of the pieces of input would have resulted in an output, that piece is displayed as the next member of the algorithmic list (and some kind of phantom steps are continued with it so as to make the above description with " $n-1$" , "$n-2$" etc. accurate—of course, we could have just made it "137" steps each time, instead of only one!).

See the discussion of recursive enumerability in Subsection IV-5 for the mathematical versions of the informal arguments just above, in particular, the proof of **IV-5.1**.

From now on, we shall mostly concentrate on algorithms as in (ii). But first, here is an example of what we were discussing just above.

There are lots of algorithms for "spitting out" the decimal digits of π : $3, 1, 4, 1, 5, \cdots$. For example, a very slow and simple-minded one could use

the infinite series

$$\pi \;=\; 4(1 \;-\; 1/3 \;+\; 1/5 \;-\; 1/7 \;+\; 1/9 - \cdots) \;\; .$$

On the other hand, it is not at all obvious whether there is an algorithm to decide the following question:

For which n is it true that, in π's decimal expansion, a consecutive block consisting of "n" consecutive copies of n (its decimal system name) occurs?

For example, when $n = 12$, we're asking whether a consecutive sequence of "24" digits, 121212121212121212121212, ever occurs. On the one hand, such finite sequences seem too regular to occur very often in such an apparently 'random' infinite sequence as π's digits. On the other hand, surely if it's random, *any* finite sequence eventually appears. In any case, a mathematical proof one way or the other for a given n, much less a method for settling it for all n, is not so obvious.

But is it not? Why not just inspect the sequence of digits as they come out of the 'machine' above? This will certainly eventually give the answer "yes" for those n for which that is the answer. But for those for which "no" is the answer, you'd never find out by this so-called "British museum search" method.

What we have just been explaining may be summarized as follows. Let's keep things 'numerical', and use 1 for "yes" and 0 for "no". What we said above is that it is not at all clear whether the total function f, where

$$f(n) := \begin{cases} 1 & \text{if "}n\text{" consecutive } n'\text{s do appear ;} \\ 0 & \text{if not ,} \end{cases}$$

is computable. Whereas the partial function g, where

$$g(n) := \begin{cases} 1 & \text{if "}n\text{" consecutive } n'\text{s do appear ;} \\ \text{undefined} & \text{if not ,} \end{cases}$$

certainly *is* computable.

(A partial function is a bit like calling out to your partner, not too loudly, in a possibly empty house, "Are you still in the house?")

One "spitting" algorithm, corresponding to the partial function g just above, would make a list, producing one copy of "n", each time a consecutive occurrence appeared. Certainly 1 would appear in the list, and I imagine 2

also would since surely you needn't look too far in π's expansion to find a pair of consecutive 2's, but three 3's??

If you just had a mechanism which produced this latter sequence of n's, without knowing what they related to, you could clearly attach an input device to it, which took in any single natural number, and a 'comparer device', which waited until (and if!) that input appeared in the sequence, at which point a 1 is produced and the mechanism halts. That's the easier half of the connection between (i) and (ii) above.

On the other hand, if you just had a mechanism which computed g, without knowing what it related to, you could produce a (possibly finite) sequence listing all the n's for which the g mechanism produced an answer, by putting 1 into it and doing "137" steps, then 2 and the first "137" steps for it, then the next "137" steps for 1, then the first "137" steps for 3, the second "137" steps for 2, the third "137" steps for 1, then the first "137" steps for input 4, and so on. Each time the g mechanism halted and wanted to output its answer 1, you add the input that caused this to the sequence you are producing. That's the less easy half of the connection between (i) and (ii) above.

Notice that in both paragraphs above, there's no need for going back and inspecting the ongoing sequence of π's digits, which gave us the problem to begin with.

The general theory of this sort of situation may be found in subsections IV-1 , particularly in IV-5, and in V-2.

It is important to note that one can have a proof that a function is computable long before the time, if ever, that a specific algorithm for computing it is discovered. In fact, the function could be simply a *constant* function. For example, for every n, let

$$f(n) := \begin{cases} 1 & \text{if } ***** \text{ is true ;} \\ 0 & \text{if } ***** \text{ is false ,} \end{cases}$$

where $*****$ is some perfectly definite, but as yet unproved, mathematical statement. To be specific in 2005, take $*****$ to be Goldbach's conjecture, that every even integer larger than 2 is the sum of a pair of primes.

The word "proof" in the paragraph above does not mean "proof acceptable to an intuitionist". However it's more the matter of whether the above function is even *well-defined* that an intuitionist would question here.

Before going on to several *mathematical* definitions of computable in the next section, there are two points with respect to possibility (ii) above which should be discussed. One visualizes the situation in (ii) (with an input which isn't in the function's domain) as happening in the computing device as a kind of 'infinite loop' (or, more accurately, as 'looping infinitely'). For example, those having some acquaintance with computer languages (alternatively, read the definition of "Babbage computable" at the end of Section II below) will recognize the infinite loop in the following program:

while $0 < x$
do $x \Leftarrow x + 1$

This computes the function mapping 0 to 0, but 'does an infinite loop' when any $x > 0$ is the input number.

The first point concerning (ii) is this. Some may initially object to the requirement that the list of instructions be finite. They might advocate that an unending list of instructions could be used as follows. The idea would be that any input which is in the function's domain would lead to only finitely many of the instructions being carried out before termination. But, as well as possibly infinite loops, some input, not in the domain, might result in the machine 'marching to infinity', that is, carrying out instructions further and further out in the list, with never a termination. It is true that one can sometimes concoct a 'credible' infinite list of instructions as above for a function which is computable in the sense of the next section. But it always turns out that there is then a *finite* list of 'instructions for generating that infinite instruction list'. And so our criterion of finiteness is in fact satisfied. As you will see below, a very useful and convincing theory of computability exists which results in most functions $\mathbf{N} \to \mathbf{N}$ ("most" in the sense of cardinality) not being computable. However, if some sort of infinite instruction lists as above were to be allowed, it is hard to see how one would exclude something like the following (non-credible) infinite list of instructions (cannot use "incredible"—that word has lost its meaning, or at least its credibility!):

if $x \approx 0$
then $x \leftidx{\div}{\leftarrow} f(0)$
else **if** $x \approx 1$
then $x \leftidx{\div}{\leftarrow} f(1)$
else **if** $x \approx 2$
then $x \leftidx{\div}{\leftarrow} f(2)$
else
etc.$\cdots\cdots$

This would apparently compute any given $f : \mathbf{N} \to \mathbf{N}$, including all the uncountably many f which are non-computable in the sense of the theory below, a theory whose acceptance is near universal. So henceforth, *finite* lists of instructions will be de rigeur, or at least their analogues appropriate to the situation—*finite* λ-expressions, *finite* Turing machine state sets, etc.

The second point concerning (ii) is a very major one, whose lack of recognition, in historical retrospect, was a major hurdle in the way of understanding computability in its present, presumably final, sense.

Diagonalize, diagonalize,
Give up everything you prize!

Cantor's Cardinality Curse on 19th Century Constructivists
See [**MeMeMe**], pp. 73-74 .

We continue this discussion in a very informal manner. However informal, the following argument seems impossible to counter, despite not being expressed in precise mathematical language. [It summarizes the dilemma, from the latter half of the 19th century, into which Cantor landed Kronecker and his followers with their insistence on some (as of then imprecise) form of finiteness/computability for functions from **N** to **N**, and in particular for the description of real numbers. In comments on the history of mathematics, that milieu is often somewhat misrepresented as Kronecker on the offensive and Cantor on the defensive, whereas the situation was more the opposite, as Webb in [**MeMeMe**], pp. 73-4 makes rather clear.]

It would appear that any theory, supposedly capturing **all** computable **total** functions $f : \mathbf{N} \to \mathbf{N}$, would necessarily involve the following features. For each such f, there would be some (undoubtedly many) algorithms which compute f. Every one of these algorithms would be written as a finite list

of instructions in a language whose allowable symbols and whose 'rules of formation of instructions' had been completely specified beforehand. But then a mechanical procedure would exist which 'recognized' whether or not a finite string of symbols *was* actually a well-formed list of instructions in that language. From this, one could surely concoct a mechanical procedure (as in possibility (i) above) which had no input, but step-by-step, made an exhaustive list of all algorithms for our computable total functions above. This only requires that the set of symbols used in the language is itself algorithmically listable, that instructions are some of the finite lists of these symbols, and that the above mentioned procedure exists (which decides which such finite lists of symbols are actually lists of instructions). But now, if the ith instruction list (in this algorithmically produced list of instruction lists) produces the computable function f_i , consider the function

$$d : \mathbf{N} \to \mathbf{N} \quad \text{defined by} \quad d(i) := f_i(i) + 1 \quad \text{for all} \quad i \in \mathbf{N} \ .$$

It is clear that d can be computed by a mechanical procedure: Given the input i, run the 'instruction list lister algorithm' till it has produced its ith instruction list; then apply that ith instruction list to the input i; and finally add 1. However, it is even easier to see that $d \neq f_i$ for any i, since those two functions don't agree when evaluated at i.

This contradicts the original assumption that one had a theory of all computable functions from $\mathbf{N}$ to $\mathbf{N}$. (See Theorem **IV-1.7** below for the mathematical version of this argument.) But the same attempted argument would *not* contradict the possibility of a theory of all computable functions from [subsets of $\mathbf{N}$] to $\mathbf{N}$. The argument given would just produce a particular computable function that is not total (but which particular one would depend on the details of the "algorithm list lister algorithm").

The contradiction could be made to arise only if we assumed also that *there were an algorithm which, given a list of instructions to compute f, would allow one to decide about membership in the domain of f*. The four theories mentioned below do not have that feature, and of course we see now that no such theory could. There will be a mechanism as in (i) to *list* the domain of f, but not necessarily one to *decide membership* in the domain of f. (See Theorem **IV-1.5** below for a mathematical version of this.) This recalls the discussion in Chapter 8 of [**LM**] about Church's negative solution of Hilbert's "Entscheidungsproblem". He showed that no algorithm existed

to decide deducibility in 1$^{\underline{\text{st}}}$order number theory (and so in mathematics generally), despite one in principle being able to algorithmically list all the deducible formulae.

The essential point from the last several paragraphs was really only well understood sometime after 1930, and had surely been a major roadblock.

For readers with some knowledge of Cantor's theory of cardinal numbers, note that a much cruder version of the argument previous quickly shows that non-computable functions must in a sense be much more abundant than computable ones. The existence of a listing as indicated in that argument (quite independently of the algorithmic nature of that listing) shows that the set of computable functions will necessarily be a countable infinite set. But it is well known (again by Cantor's diagonal argument) that the set of all functions (even just from **N** to itself, much less partial functions of possibly more than one variable) is an uncountable infinite set.

Let's move on to the various mathematical definitions of computability.

II. Mathematical Definitions of Computability

There have been many (equivalent) such definitions in the past 70 years. In some instances, particularly Turing's invention of what came to be called Turing machines, fairly persuasive arguments were also given that the definition proposed does in fact exhaust all possibilities for computation by mechanical procedures. On the other hand, serious counterarguments to this claim of exhaustiveness are occasionally presented by other logicians. Furthermore, philosophical disputes have arisen as to whether Turing's arguments determine the limit of what *anything* can do (including the human mind), or 'merely' what a 'machine' can possibly do. Actually, Turing phrased his arguments initially in the form of thinking about a *human* carrying out a purely clerical routine, so the argument can sometimes go the other way as well. Here we shall pass over all this and simply accept the Church-Turing Thesis: *that no function is computable unless it can be computed by one (and hence all) of the definitions below.* (See [**MeMeMe**] for a very deep historical discussion by Webb of most of this. I stole the above label "Cantor's ... Curse ..." from him.)

It is fair to say that most mathematical definitions of computability (certainly the three other than Turing's below) initially strike one with the following thought : "Surely one can invent a computable function which does not fall under the aegis of this definition!" One of the main sources of empirical/historical evidence for the 'correctness' of such definitions is that (so far!) the outcome of this has always turned out to be negative. In particular, mathematical proofs can be found showing that the set of functions computable in the sense of each of the other three definitions just below is exactly the same as the set of Turing machine-computable functions. We shall present one proof along these lines in Sections III and IV.

Our four definitions will be:

(1) λ-definability, producing the $\mathcal{LC}$-functions;

(2) recursivity, producing the $\mathcal{RC}$-functions;

(3) Turing computability, producing the $\mathcal{TC}$-functions;

(4) Babbage computability, producing the $\mathcal{BC}$-functions.

The last of these is what may possibly be a small contribution to the pedagogy of this subject. See the latter parts of this section. There will actually be a fifth, quite different, but equivalent, definition, in Subsection V-3, namely

(5) $1^{\underline{\text{st}}}$order computability.

But don't get your hopes up : a consequence of Gödel incompleteness, and the Grassman-Dedekind-Peano 2nd order characterization of **N**, is that there can be no 'decent' proof system for higher order logic, so a higher order computability is apparently out of the question.

There will even be a sixth, equivalent, definition, in Subsection IV-11, namely

(6) **McSELF**-computability,

which is based on an elegant language due to McCarthy, which puts heavy emphasis on self-referential (also called recursive) commands.

The first three (with minor variances of detail) are the historically first definitions, given respectively by Church-Kleene, by Gödel-Herbrand (along with a formal 'equation calculus'—see [**M**]), and by Turing, in the period 1932–36.

As for (1), details are delayed to Section VII. There we use the opportunity to reinforce the general idea of a *formal system*, by presenting one such system, the λ-*calculus*, initially in a totally unmotivated way in order to emphasize the 'formality'. For a quick taste of this : each computable function corresponds to at least one formal expression, looking something like the following example:

$$\lambda f \bullet \lambda g \bullet \lambda x \bullet \lambda y \bullet ((fx)((gx)y)) \quad .$$

Once the motivation has been given for the λ-calculus, each such formal expression can be conceived as giving a sort of 'pre-machine language' string, which produces algorithms to compute that function, one algorithm for each choice of which order to 'reduce', using λ-calculus rules, the expression or 'λ-term' which is the above string followed by two strings representing the input integers. For example, we'll see later that the displayed λ-term above may be thought as computationally defining $(u, v) \mapsto u + v$, the addition function of two variables (or, more accurately, the function of u whose value is the function of v which adds u to v, that is, $u \mapsto [v \mapsto u+v]$). The λ-calculus has become more the domain of computer scientists than of mathematicians in recent years, especially in the study of so-called **functional programming languages** (some well known examples being LISP, SCHEME, PERL and S).

As for (2), the definition of *recursive function* is given near the beginning of Appendix A in Section IV, and also in [**LM**], p.403. It is essentially

repeated in the next section. Note that, modulo accepting the 'obvious' fact that the initial functions (addition, multiplication, projection, characteristic function of $<$) are all computable, each 'recursive verification column' may be conceived as a computational algorithm for the function at the bottom of that column. Each succeeding line in such a column would be a function which is either initial, or is explicitly defined using composition or minimization involving functions from lines higher up.

Sketch on Turing machines.

As for (3), the definition is a few paragraphs below, after some further heuristic discussion.

In theoretical CS, a universal Turing machine is the ubiquitous model for an actual computer. Such a device differs from a real computer in at least three respects:

(a) it is staggeringly less efficient in the practical sense for all the kinds of computing that practical people want done;

(b) it has no upper limit on the amount of memory that can be used;

(c) it is an abstract mathematical object, not a physical object (though it models a physical object); and so it is completely error-free.

If the Turing machine version of the Church-Turing thesis is accepted, then one can conclude that any conceivable computer could be thought of as doing no more nor less than this : It mechanically applies itself, step-by-step, to the conversion of any finite arrangement of check-marks, $\surd$, to produce other such arrangements, one for each step; and it 'knows' when, if at all, to stop making these steps. Such a finite arrangement of $\surd$'s we can take to be along a horizontal line, and we can even reduce to the case where the $\surd$'s only appear singly or in pairs, and the spaces between them also appear singly or in pairs (except for the infinite spaces off to the left and right of the array).

For example,

$$\cdots\square\square\square\surd\surd\square\surd\square\surd\surd\square\square\surd\surd\square\surd\surd\square\surd\square\surd\square\square\surd\surd\square\surd\square\square\square\cdots\cdots$$

could be thought of as corresponding to (5, 12, 2) via its binary representation (101, 1100, 10); so a single $\surd$ corresponds to the digit 0, a double $\surd\surd$ to 1, and a double space indicates the start of the next entry.

Though inefficient, a theoretically perfectly good alternative to the above coding conventions would be to use simply a string of "$n+1$" $\surd$ symbols to represent the natural number n, and a single blank to separate entries from each other. So in this scheme, the quadruple (5, 12, 2, 0) would be represented as

$$\cdots\square\square\square\surd\surd\surd\surd\surd\surd\square\surd\surd\surd\surd\surd\surd\surd\surd\surd\surd\surd\surd\surd\square\surd\surd\surd\square\surd\square\square\square\cdots\cdots$$

Actually, below we shall allow as our set of symbols any finite set containing the blank symbol $\square$ plus at least one other symbol. We won't be concerned about which scheme of coding might be chosen to represent natural numbers or anything else. A Turing machine will be allowed any such set of symbols. Its 'operation' will define a (partial) function from the set of finite strings of symbols to itself. Since we won't be going too seriously into the theory of Turing machines, there will be no need to be more specific.

The 'physical' picture of a Turing machine is as follows: it consists of

(a) a 2-way infinite tape divided into indistinguishable squares (or, a finite tape with one little man at each end who has a large supply of duct tape etc., in order to add extra squares as needed during a computation);

(b) each square contains a symbol (see the previous two displays for examples);

(c) the tape is being scanned one square at a time by a read-write head, which reads the symbol on that square, and then, depending on its *state* and which symbol it has just read, it does three things:

(i) It erases what it read and then, on the same square, it writes a new symbol (including possibly the same symbol it just erased or possibly the *blank symbol* $\square$).

(ii) It moves one square to the left, or not at all, or moves one square to the right.

(iii) It goes into a new state, including possibly the same one it was in, or possibly the *end* state e, or even possibly returning to the *begin* state b.

In the definitions below, we shall be regarding the computation as having finished if and when the machine enters the end state. Thus what the machine would do as the next step, when its state happens to be e, is actually irrelevant to the behaviour we are interested in. But, for the sake of symmetry, the transition behaviour of the machine when in that end state will be

included with all the other transition behaviours, which are the real 'guts', namely the function T, in the following definition.

A point which sometimes causes confusion is that states are not something whose 'internals' are known. For a Turing machine, a state is really just the 'name for a state'. It's only when you're in the business of actually building a physical device that you need to worry about how the state does what it does according to the function T.

Definition. A (deterministic) **Turing machine** $\mathcal{T}$ is any triple

$$(\ (St, b, e) \ , \ (Sy, \square) \ , \ T \ ) \quad .$$

where St and Sy are disjoint finite sets, whose elements are to be called *states* and *symbols* respectively, with the following two properties:

(i) T is a function

$$St \times Sy \ \longrightarrow \ St \times Sy \times \{L, N, R\} \ .$$

(ii) $\square \in Sy \quad , \quad e \in St \quad \text{and} \quad b \in St \ .$

(The graph of T therefore consists of some ordered pairs $((t_1, y_1), (t_2, y_2, M))$, which can be read as quintuples, that often being the way a Turing machine is described in popular treatments.)

Here $t_i \in St$, $y_i \in Sy$ and $M \in \{L, N, R\}$. The elements of that last set could have longer names, viz. *Left, Nomove, Right.* The function T is what governs the transition behaviour of the machine, as described roughly just before the definition, and mathematically immediately below.

Definition. A *computation* of the Turing machine $\mathcal{T}$ is a (possibly unending) sequence $\mathcal{E} \ = \ (E_1, E_2, E_3, \cdots)$, where each *environment* E_i is a 2-way infinite sequence

$$\cdots, \square, \square, \square, y', y'', \cdots, t, y, \bar{y}, \bar{\bar{y}}, \cdots, \square, \square, \square, \cdots$$

(we are thinking of what's written on the tape at a given time, with one state symbol t inserted, to indicate both which state the machine is in and the fact that the head is reading the square just after where t occurs) such that the following hold:

(i) All but finitely many terms in each environment are the blank symbol. (Many might prefer to regard each environment rather as a finite sequence, what we'll call

the *unblanking* of the environment, removing all blank symbols except for those having a non-blank both somewhere to the right and somewhere to the left.)

(ii) All but one of the terms are symbols, the exceptional term being a state.

(iii) The (unique) state term in E_1 is b.

(iv) The final state e occurs as the (unique) state term in some environment of $\mathcal{E}$ iff that environment is the last environment of $\mathcal{E}$ (which may not exist).

(v) For successive environments E_i and E_{i+1}, if we write E_i as (C, y', t, y, D), where t is its unique state term, and C and D are sequences which are 1-way infinite to the left and to the right respectively, and if $T(t, y) = (\bar{t}, \bar{y}, M)$ (where T is the 'function component' of the Turing machine $\mathcal{T}$), then

$$E_{i+1} \;=\; \begin{cases} (C, \bar{t}, y', \bar{y}, D) & \text{if } M = L \ ; \\ (C, y', \bar{t}, \bar{y}, D) & \text{if } M = N \ ; \\ (C, y', \bar{y}, \bar{t}, D) & \text{if } M = R \ . \end{cases}$$

(This last condition is the real 'meat', writing down just what the transitions will be. It is clear that, once E_1 is given, the entire sequence $\mathcal{E}$ is determined, for a given Turing machine. A point which might (but shouldn't) bother mathematicians is that we've ignored the parametrization of environments. That is, a 2-way sequence would often be defined to be a function with domain $\mathbf{Z}$. But we are identifying two such functions if they differ by a translation of $\mathbf{Z}$, since everything we discuss is invariant under such re-parametrizations. So we'll treat environments in this more informal, picturesque manner.)

Note that the end state e occurs not at all in an unending computation $\mathcal{E}$, and exactly once, in the last environment, in a finite computation $\mathcal{E}$. A computation *terminates* iff it is finite, and then it *terminates with* that last environment.

Now it is convenient to temporarily introduce a brand new symbol err, to have a way of denoting an unending computation. Let Sy^* denote the set of all finite sequences of symbols.

Definition. The Turing machine $\mathcal{T}$ defines the function

$$||\mathcal{T}|| \ : \ Sy^* \cup \{err\} \ \rightarrow \ Sy^* \cup \{err\} \quad \text{by}$$

$$err \mapsto err \qquad \text{and}$$

$$(y_1, \cdots, y_k) \mapsto \begin{cases} err & \text{if the computation starting with } \cdots, \square, \square, b, y_1, \cdots, y_k, \square, \square, \cdots \text{ doesn't terminate;} \\ (y'_1, \cdots, y'_\ell) & \text{if that computation terminates with an environment whose unblanking with } e \text{ removed is } (y'_1, \cdots, y'_\ell) \ . \end{cases}$$

Finally, the partial function on finite strings of symbols defined by $\mathcal{T}$ will be denoted $|\mathcal{T}| : D \rightarrow Sy^*$. It is the restriction of $||\mathcal{T}||$ to the domain

$$D \ := \ Sy^* \ \setminus \ ||\mathcal{T}||^{-1}\{err\} \quad .$$

It is this last function, for various choices of Turing machine, which gives the definition of just what is **Turing computable** for functions of symbol strings. To do so with functions of natural numbers just amounts to making a choice of coding, as illustrated above, and possibly also making a choice of which portion of the final environment is to be regarded as the actual output. The endproduct, the set of $\mathcal{TC}$-functions, is independent of what those latter choices are, as long as they are moderately sensible.

Just below are a few of many possible technical variations on this definition, variations which don't affect the set of $\mathcal{TC}$-functions. Some of these may make the definition easier to work with, which we won't be doing.

(i) Sometimes the set of symbols Sy is fixed once and for all beforehand for all Turing machines, and it can be as small as two elements, for example $\{\surd, \square\}$.

(ii) There are versions in which the tape is 1-way infinite.

(iii) It is possible to dispense with the end state, and have T defined only on a *subset* of $St \times Sy$, regarding a computation as terminating if and when it arrives at an environment $\cdots\cdots, t, y, \cdots\cdots$ for which $(t, y) \notin \text{Domain}(T)$. It is straightforward to see how to go back-and-forth between our setup and

this one. Clearly, in our setup, the values $T(e, y)$ are of no interest for any y, and, if $T(t, y) = (e, y', M)$, then the value of M is irrelevant.

Although we shall not be dwelling on Turing machines, there should be one specific example here. The one below is a bit interesting because, for a later language which we call **ATEN** (and which can also serve as a basis for defining computability) the analogous computation is surprisingly hard to achieve (in a certain sense).

Here is a Turing machine which will switch the entry initially being read with the one immediately to its right, return the head to its original position, and then terminate the computation, *leaving all other squares as is*. The reader should work out this behaviour to see how the tiny idea for setting this up arises.

The elements listed below are assumed to be all different from each other:

$$Sy := \{ \ \square \ , \ \surd \ \} \quad ; \quad St := \{ \ b \ , \ c_0 \ , \ c_1 \ , \ d_0 \ , \ d_1 \ , \ e \ \} \ .$$

The function T is specified by the following 'quintuples', where we use "$-$" to indicate that it doesn't matter what you choose for that particular slot.

$$\begin{array}{lll} (b, \square) \ \mapsto \ (c_0, -, R) & ; & (b, \surd) \ \mapsto \ (c_1, -, R) \ ; \\ (c_0, \square) \ \mapsto \ (d_0, \square, L) & ; & (c_0, \surd) \ \mapsto \ (d_1, \square, L) \ ; \\ (c_1, \square) \ \mapsto \ (d_0, \surd, L) & ; & (c_1, \surd) \ \mapsto \ (d_1, \surd, L) \ ; \\ (d_0, -) \ \mapsto \ (e, \square, N) & ; & (d_1, -) \ \mapsto \ (e, \surd, N) \end{array}$$

$$(e, -) \ \mapsto \ (-, -, -) \ .$$

Before going on to the fourth definition of computability, $\mathcal{BC}$-functions, here are some remarks on the earlier claim that all four definitions are equivalent, in the sense that which functions turn out to be computable is the same every time. The next two sections will contain the proofs of one case of this, namely :

in III, that every $\mathcal{RC}$-function is a $\mathcal{BC}$-function;
in IV, that every $\mathcal{BC}$-function is an $\mathcal{RC}$-function.

As for the three definitions already discussed briefly above, their equivalence was quickly established in the 1930's by the pioneers of the subject (and, as

mentioned, this was important evidence for the acceptability of the Church-Turing thesis). Since [**D-S-W**] for example, has discussions of many of these equivalence proofs, and I cannot see how to improve on them, we'll not give any of these details here. See also [**S**] and [**M**] for several very nicely organized such proofs, in the terse S&M style appropriate to educating already somewhat accomplished mathematicians.

Babbage computability, BTEN language, and $\mathcal{BC}$ functions.

Many students have some acquaintance with high level computer languages such as PASCAL and C++. The definition below amounts taking a few basic commands common to these sorts of languages to produce a simple language **BTEN**. This is used to mathematically define *Babbage computable* by taking advantage of the fact that readers here are by now assumed to be fairly well acquainted with [**LM**]. Even slight familiarity with $1^{\underline{\text{st}}}$order number theory and its 'formal' semantics (Tarski definitions in Chapter 6 of [**LM**]) will make it easy to define exactly which functions **BTEN** does compute.

Symbols for BTEN

$$\| \text{œ} \quad \textbf{ite} \quad \| \quad \textbf{whdo} \quad \| \quad \dot{\leftarrow} \quad \| \quad ; \quad \| \quad) \quad \| \quad (\quad \| \quad \mathcal{E} \quad \|$$

$$\| \quad \mathcal{B}_{i_1,i_2,\ldots,i_k} \text{ , where } i_1 < i_2 < \ldots < i_k \text{ are natural numbers and } k > 0 \quad \|$$

$$\| \quad \text{all the symbols of the assertion language just below,}$$

$$\text{including variables } x_0, x_1, x_2, \cdots \quad \|$$

Assertion Language

This is $1^{\underline{\text{st}}}$order number theory, but without $\forall$ and $\exists$, and with the variables indexed as above, starting with 0 rather than 1. (See [**LM**], Section 5.1 and Chapter 6.) Thus the 'slot vectors' $\underline{v}$ in the semantics should also start with 0, i.e. $(v_0, v_1, v_2, \cdots)$. Here we shall almost always be using $\mathbf{N}$, the set of natural numbers, as the interpretation, so each $v_i \in \mathbf{N}$. The assertion language consists then of the *quantifier-free* formulae from the language of $1^{\underline{\text{st}}}$order number theory, and the terms from that language.

Command Language BTEN

The set of commands is defined inductively as the smallest set of finite strings of symbols for which (1) to (5) below hold :

(1) $\boxed{x_i \dot{\leftarrow} t}$ is a command for each variable x_i and each term t in the assertion language. (The 'phrase' $x_i := t$ is possibly more customary for this command, but we wish to reserve := for "is mathematically defined to mean".)

(2) $\boxed{\textbf{ite}(F)(C)(D)}$ is a command, for each (quantifier-free) formula F in the assertion language, and all commands C and D.

This will always be 'abbreviated' to

if F
thendo C
elsedo D

[Perhaps overly pedantic, and the words "thendo" and "elsedo" are ugly, but we prefer not to confuse the if-then-else connective in logic, which builds assertions, not commands, with the **ite** command. After all, the following abbreviated command using an abbreviated $1^{\underline{\text{st}}}$order formula :

if if $x < 1$ then $y = z$ else $y < z$
thendo C
elsedo D ,

when communicated orally, would cause some initial confusion to the most experienced of us, much less a beginner, were we to drop the **dodo**'s.]

(3) $\boxed{\textbf{whdo}(F)(C)}$ is a command, for each (quantifier-free) formula F in the assertion language, and all commands C.

This will always be 'abbreviated' to

while F
do C

(4) $\boxed{\mathcal{B}_{i_1,i_2,\ldots,i_k}\ C\ \mathcal{E}}$ is a command, for any command C.

(5) $\boxed{(C_1; C_2; \cdots; C_k)}$ is a command, for any $k \geq 2$ and commands C_i.

By specifying the semantics of **BTEN**, we simply mean specifying what each command 'actually does'. We conceive of the abstract machine here as having a sequence of 'bins' (or 'memory locations', if you prefer), each one containing a natural number, say

$$v_0 \text{ in bin } 0 \ , \ v_1 \text{ in bin } 1 \ , \ v_2 \text{ in bin } 2 \ , \ \ldots\ldots$$

'Doing' a command changes what is in some of the bins. As is easily seen from the definitions below, a command will actually change only finitely many bins. Philosophically, perhaps it would be safer also to say that all but finitely many bins contain 0. That situation might be pictured as having mostly empty bins. This aspect plays no particular role below, so will not be dwelt on further. But see Subsection IV-2.

An important point is that the **while-do** commands will sometimes produce the "infinite loops" referred to when we earlier discussed the need to consider non-total functions. For this, it's convenient to introduce an object with the name *err*. That can be any mathematical object which is not a member of

$$\mathbf{N}^{\infty} \ := \ \{ \ (v_0, v_1, v_2, \cdots) \mid v_i \in \mathbf{N} \ \} \ .$$

We'll define, for each command C, a function $||C||$. By convention, functions always map *err* to itself (" errors propogate" or, "basepoints are preserved"). The need for *err* is, as noted above, that a **while-do** command C will give a function $||C||$ which may map some elements of $\mathbf{N}^{\infty}$ to *err*. In general, the function $||C||$ captures mathematically what the command C 'actually does to the numbers in the bins'. I'll assume that no further motivation is needed for the definitions below of the semantics of **if-thendo-elsedo** or **while-do** commands.

The other three types of commands behave intuitively as follows :

The " $x_i \div\!\!= t$ " command only alters what is in bin_i . It first takes the given contents of the bins and calculates, from the term t, the natural number $t^{\underline{v}}$. Then it puts that number in bin_i (after removing what was there), and leaves the other bins unaltered. For example, if $t = (1 + x_7) \times (x_3 + 0)$, then after the command is executed, bin_i contains $(v_3 +_{\mathbf{N}} v_7 v_3)$ in place of v_i , where v_3, v_7 and v_i were in bins 3 , 7 and i beforehand. We have fussily used $+_{\mathbf{N}}$ to denote the actual operation, to distinguish it from the corresponding symbol " + " in the assertion language, (and also have used juxtaposition for actual multiplication, of course).

The " $\mathcal{B}_{i_1,i_2,\ldots,i_k} \ C \ \mathcal{E}$ " command does C , but then, for $1 \leq \ell \leq k$, restores to each bin_{i_ℓ} the number which was there before the execution started. $\mathcal{B}$ is for "**begin**", and $\mathcal{E}$ is for "**end**".

The " $(C_1; C_2; \cdots; C_k)$ " command first does C_1, then does C_2, ..., finally,

does C_k.

After all this motivation, here are the equalities which mathematically define the semantics. The definition is by 'induction on commands', so, except for the first one, we are assuming inductively that $||C||$ has been defined for each of the ingredient commands C used to build up the command whose semantics is being specified. This definition produces, for every command C, a function $||C|| : S \to S$, where $S := \mathbf{N}^\infty \cup \{err\}$. Recall that every function maps err to itself, so we don't repeat that below. We need to specify the effect of $||C||$ on each $(v_0, v_1, v_2, \cdots) = \underline{v} \in \mathbf{N}^\infty$.

Definition of the semantics of BTEN .

$$||x_i \Leftarrow t||(v_0, v_1, v_2, \cdots) := (v_0, v_1, \cdots, v_{i-1}, t^{\underline{v}}, v_{i+1}, \cdots) \quad \text{(see [LM], p.211).}$$

$$||\mathbf{ite}(F)(C)(D)||(\underline{v}) := \begin{cases} ||C||(\underline{v}) & \text{if } F \text{ is true at } \underline{v} \text{ from } \mathbf{N} ; \\ ||D||(\underline{v}) & \text{if } F \text{ is false at } \underline{v} \text{ from } \mathbf{N} . \end{cases}$$

$$||\mathbf{whdo}(F)(C)||(\underline{v}) := \begin{cases} \underline{v} & \text{if } F \text{ is false at } \underline{v} ; \\ err & \text{if } F \text{ is true at } ||C||^k(\underline{v}) \text{ for all } k \geq 0 ; \\ ||C||^k(\underline{v}) & \text{if } F \text{ is true at } \underline{v},\ ||C||(\underline{v}), \cdots,\ ||C||^{k-1}(\underline{v}), \\ & \quad \text{but false at } ||C||^k(\underline{v}) . \\ & \quad \text{(By convention here, } F \text{ is false at } err.) \end{cases}$$

$$[\ ||\mathcal{B}_{i_1,i_2,\ldots,i_k}\, C\, \mathcal{E}||(\underline{v})\ ]_j := \begin{cases} [||C||(\underline{v})]_j & \text{if } j \neq i_\ell \text{ for any } \ell ; \\ v_j & \text{if } j = i_\ell \text{ with } 1 \leq \ell \leq k . \end{cases}$$

$$||(C_1; C_2; \cdots; C_k)||(\underline{v}) := ||C_k||(||C_{k-1}||(\cdots(||C_1||(\underline{v}))\cdots)) .$$

Remark. It should be stated emphatically that the function $||C||$, in cases where it maps some number sequences to err (that is, the command C involves some **while-do** command which sometimes causes an infinite loop) is not to be regarded as 'computable' on its entire domain. Expressed another way, it is definitely false that every recursive function is the restriction to its own domain of a *total* recursive function. One can extend the function in some arbitrary way outside its domain to produce a total function. But even

just defining it to be constant on the complement of its domain will normally *not* result in a *recursive* total function. This can be established in general by the same diagonal argument as occurs in the latter part of section I (Cantor's Curse). It is true that one can extend the domain of a non-total recursive function by 'unioning' that domain with any *finite* subset of its complement, then extend the function to be constant on that finite set, and the result is still recursive. But recursive functions whose domain-complements are not recursively enumerable will often not be extendable to total recursive functions. (For a specific example, see Turing's unsolvability of the halting problem, **IV-1.5**, and contrast that theorem with the partial computable function displayed just prior to the theorem statement. See also Subsection IV-5, though that is one of several extras which are not crucial to our main purpose here.)

Definition of $\mathcal{BC}$-computable. If $f : D \to \mathbf{N}$ is a function, where $D \subset \mathbf{N}^k$ for some $k > 0$, we shall say that f is *Babbage computable* (or simply $\mathcal{BC}$) if and only if there is a command C in **BTEN** such that, for all $(v_1, \cdots, v_k) = \vec{v} \in \mathbf{N}^k$, there are natural numbers $w_1, w_2, \cdots$ such that

$$||C||(0, \vec{v}, 0, 0, \cdots) = \begin{cases} err & \text{if } \vec{v} \notin D \ ; \\ (f(\vec{v}), w_1, w_2, \cdots\ ) & \text{if } \vec{v} \in D \ . \end{cases}$$

This just says that computability of f is equivalent to the existence of a command which, when applied with zeroes in all the bins except bins 1 to k, will always try to calculate the value of f applied to the contents of bins 1 to k (in that order) and place the answer in bin zero; but it will produce *err* if the k-tuple of entries in bins 1 to k is not in the domain of f (otherwise it will succeed!).

Another way to express this is to imagine starting from a command C, and asking which function of "k" variables it computes. The answer is f, where

$$f(v_1, \cdots, v_k) \ = \ [||C||(0, v_1, \cdots, v_k, \underline{0})]_0 \quad .$$

Of course we interpret the right-hand side as undefined if it is an attempt to find the zeroth slot in *err* !

A first comment here is that there is no real need for the assumption about having zeroes outside bins 1 to k. That is, if there is a command which

computes f in the defined sense, then there is a command which computes f in the stronger sense: just take the given command and precede it by a bunch of $(x_j \dot{\leftarrow} 0)$'s, one for each j not between 1 and k for which x_j appears in the given command.

Exercise. Show that the $\mathcal{B}_{i_1,i_2,\ldots,i_k} \cdots \mathcal{E}$-command construction is redundant in the following sense. For any command C in **BTEN** and for any $\ell > 0$, there is a command A with no symbols $\mathcal{B}, \mathcal{E}$ in it, and such that the function of "ℓ" variables which A computes is the same as the one which C computes.

(Try to do this without explicitly using any **ite**-commands. That will be handy later. This exercise is not hard, but also not necessarily as easy as it first appears to be. See Subsection IV-2.)

Despite the exercise, because the $\mathcal{B} \cdots \mathcal{E}$ commands are very convenient in some of the considerations in later sections, we'll leave them in the language.

A few other comments on this definition are worth making.

First note that the assertion language formulas, since they don't involve quantifiers, can be 'truth evaluated' at any $\underline{v}$ in finitely many mechanical steps. I shall assume that the reader isn't in the mood to dispute this. Really, one needs to discuss how natural numbers are to be denoted using strings from some finite alphabet of symbols, such as briefly indicated in the discussion of Turing machines, or perhaps using binary or decimal notation. And then a discussion of how to construct an adding machine, a multiplying machine and a comparing machine would take place. See Section IV-12, where we discuss more specifically 'physical' realizations of **BTEN**, including logic gates, and how they produce the little machines just above (this all being done in a rather whimsical manner).

So there is nothing dubious from the computational viewpoint in the semantics of the **if-thendo-elsedo** and **while-do** commands.

The $\mathcal{B} \cdots \mathcal{E}$ command construction should perhaps be pictured physically as being implemented by having a second sequence of 'secret' bins where one can temporarily store numbers, to be brought back later into the bins which are labelled by variables occurring in commands. In view of the exercise above, as dealt with later in Subsection IV-2, this is not a major consideration.

Since the only atomic commands are the " $x_i \dot{\leftarrow} t$ " commands, it is

clear that we are imagining a machine which need only be able to do the following:

(i) calculating values $t^{\underline{v}}$ (i.e. behaving as a \$2 pocket calculator whose subtraction and division functions might be defective);

(ii) truth evaluating as just above;

(iii) erasing and entering bin values;

(iv) persisting.

Presumably then, no one remains unconvinced of the 'mechanical effectiveness' of **BTEN**. See also the frivolous Section IV-12. For examples of **BTEN** programs (i.e. commands) which do interesting things, just keep reading.

The Babbage busy beaver is not $\mathcal{BC}$.

Here we produce (in small print, signalling that this is not central to the main points of the work) a specific function which is not computable.

Define *length* by letting $\ell[K]$ be the number of symbols in a string, K, of symbols. For example, $\ell[\ x_{17} \Leftarrow 0\ ] = 3$. Since every command contains at least one assignment command, we see that $\ell[C] \geq 3$ for all commands C in **BTEN**. And so

$$\ell[(D;E)] \geq 1+3+1+3+1 = 9 \ ,$$

and

$$\ell[\mathbf{whdo}(F)(C)] \geq 1+1+3+1+1+3+1 = 11 \ ,$$

using the fact that $\ell[F] \geq 3$ for formulas F. For example, $\ell[0 < 0] = 3$. In particular, except for a few assignment commands, all commands have length at least 5.

We'd like to consider, for each n, the maximum value at 0 of all 1-variable functions computed by commands of length at most n. Unfortunately, there are normally infinitely many such commands, because of the infinitude of variables. So it's not completely obvious that this maximum exists. The easiest way to fix this is to use *weight* instead, where we define

$$w[C] \ := \ \ell[C] + \text{sum of all subscripts of variables occurring in } C \ .$$

For example, $w[\mathbf{whdo}(x_{103} < x_3)(\ x_{17} \Leftarrow x_{77}\ )] = 11+103+3+17+77 = 211$.

It remains true that, except for a few assignment commands, all commands have weight at least 5.

Define a 1-variable total function BSC by

$$BSC(n) \ := \ \max(\ \{0\} \cup \{\ (||C||(\underline{0}))_0 \ : \ C \in \mathbf{BTEN}\ ,\ w[C] \leq n \text{ and } ||C||(\underline{0}) \neq \mathit{err}\ \}\ ) \ .$$

"B" is for "busy", and "SC" alludes to Margaret Atwood and Canadian symbolism. The inclusion of $\cdots\{0\}\ \cup\cdots$ is to avoid trying to find the maximum element in the empty set (for the first few values of n). For larger values, $BSC(n)$ is the maximum of $f(0)$ for all 1-variable functions f which have 0 in their domains and are computable by a **BTEN**

command of weight at most n. Since the sets being maximized get larger with increasing n, we see that $BSC(a) \geq BSC(b)$ when $a \geq b$. And so

$$BSC(a) < BSC(b) \implies a < b \qquad (*)$$

Theorem. (Rado) *BSC is not a $\mathcal{BC}$-function.*

So BSC gives a concrete example of a non-computable function. The pure existence of such a function follows easily from cardinality considerations, as we noted earlier.

We shall use terms "n" in the assertion language. These are syntactic versions of the natural numbers n, namely "$0_{\mathbf{N}}$" $= 0$; "$1_{\mathbf{N}}$" $= (0+1)$; "2" $= ((0+1)+1)$; etc. Inductively, "$n +_{\mathbf{N}} 1_{\mathbf{N}}$" $:=$ ("n" $+1$) . It follows easily by induction that $w[\text{“}n\text{”}] = 4n+1$.

Proof of the Theorem. Suppose, for a contradiction, that command C_0 computes the function BSC. Choose any integer s so that

$$s \;\geq\; \frac{5w[C_0]+15}{w[C_0]-4} \qquad (**)$$

(Note that C_0 is certainly not an assignment command, so the denominator in $(**)$ is positive. The choice above was of course determined *after* the calculation below. The right-hand side in $(**)$ is $5 + (35/(w[C_0]-4))$, so $s = 40$ will do perfectly well.)

Define a command D as follows:

$$D \;:=\; (\; x_1 \leftarrow \text{“}s\text{”} \times \text{“}w[C_0]\text{”} \;;\; C_0 \;;\; x_0 \leftarrow x_0 + 1 \;) \;.$$

Certainly $||D||(\underline{0}) \neq err$; in fact, since C_0 computes BSC, we get

$$(||D||(\underline{0}))_0 \;=\; BSC(s \cdot w[C_0]) + 1 \;.$$

By the definition of BSC, we see that $BSC(w[D]) \;\geq\; BSC(s \cdot w[C_0]) + 1$. And so, by $(*)$, we have

$$s \cdot w[C_0] \;<\; w[D] \;=\; \ell[D]+1 \;=\; 3 + w[\text{“}s\text{”}] + 1 + w[\text{“}w[C_0]\text{”}] + 1 + w[C_0] + 7 + 1 \;=$$

$$13 + w[C_0] + (4s+1) + (4w[C_0]+1) \;=\; 4s + 5w[C_0] + 15 \;.$$

This gives

$$s \cdot (w[C_0]-4) \;<\; 5w[C_0] + 15 \quad ,$$

contadicting $(**)$, and completing the proof.

It is amusing and instructive to note that, in the definition of D, using "$s \cdot w[C_0]$" would not have worked. The idea here, due to Rado, came from the Richard paradox. It is somewhat related to Boolos' use of Berry's paradox to give an interesting proof of the Gödel Incompleteness Theorem. See **[LM]**, Appendix VS to Ch. 5.

III. Proof that f is $\mathcal{RC}$ implies that f is $\mathcal{BC}$.

Commands which compute the initial functions in the (recursive!) definition of *recursive* are as follows:

The addition function $(v_1, v_2) \mapsto v_1 + v_2$: $\quad x_0 \leftarrow\!\!\div\, x_1 + x_2$

The multiplication function $(v_1, v_2) \mapsto v_1 v_2$: $\quad x_0 \leftarrow\!\!\div\, x_1 \times x_2$

The ith projection function, $\mathbf{N}^k \to \mathbf{N}$, sending $\vec{v}$ to v_i : $\quad x_0 \leftarrow\!\!\div\, x_i$

The function $\overline{\chi}_<$, sending (v_1, v_2) to 0 if $v_1 < v_2$, to 1 otherwise :

if $\quad x_1 < x_2$
thendo $x_0 \leftarrow\!\!\div\, 0$
elsedo $\quad x_0 \leftarrow\!\!\div\, 1$

which, recall, is an 'abbreviation' for $\quad \mathbf{ite}(x_1 < x_2)(x_0 \leftarrow\!\!\div\, 0)(x_0 \leftarrow\!\!\div\, 1)$.

Here is one of many others which do the same job. This one has the advantage (much appreciated later) of avoiding the **ite** command construction.

$x_0 \leftarrow\!\!\div\, 0$;
while $x_2 < x_1 + 1$
do $\quad (x_0 \leftarrow\!\!\div\, 1 \ ; \ x_2 \leftarrow\!\!\div\, x_1 + 1)$

The 2nd and 3rd lines are an 'abbreviation' for

$$\mathbf{whdo}(x_2 < x_1 + 1)((x_0 \leftarrow\!\!\div\, 1 \ ; \ x_2 \leftarrow\!\!\div\, x_1 + 1)) \quad .$$

The first line isn't really needed, but makes the command 'strongly' compute the function, i.e. it relieves the need to start with zeroes except in bins 1 and 2.

To complete the proof by 'induction on recursive functions', we must deal with composition and minimalization.

For composition, we are given recursive functions as follows : f of "j" variables, and $h_1, \cdots, h_j$, each of "ℓ" variables. And we define g by

$$g(v_1, \cdots, v_\ell) \ := \ f[h_1(v_1, \cdots, v_\ell), h_2(v_1, \cdots, v_\ell), \cdots, h_j(v_1, \cdots, v_\ell)] \quad ,$$

with domains as discussed in the early part of Appendix A. (See also p.403 of [**LM**].)

Since we are proceeding by induction, we assume that commands, which we'd denote $C[f]$, $C[h_1]$, $\cdots$, $C[h_j]$, all exist to compute the 'ingredient' functions. By the remarks immediately after the definition of "$\mathcal{BC}$-computable", we can assume that they "strongly compute" the relevant functions. That is, in the example $C[f]$, no matter what is in bin 0 or bins k for $k > j$, with $v_1, \cdots, v_j$ in bins 1 to j respectively before executing $C[f]$: after executing it, in bin 0 you will find $f(v_1, \cdots, v_j)$ if $(v_1, \cdots, v_j) \in$ domain(f); and the computation will be non-terminating if $(v_1, \cdots, v_j) \notin$ domain(f) .

We must find a command $C[g]$. Here it is :

$(\mathcal{B}_{1,2,\ldots,\ell}\ C[h_1]\ \mathcal{E}\ ;\ x_{\ell+1} \div\!\leftarrow x_0\ ;$

$\mathcal{B}_{1,2,\ldots,\ell+1}\ C[h_2]\ \mathcal{E}\ ;\ x_{\ell+2} \div\!\leftarrow x_0\ ;$

$\mathcal{B}_{1,2,\ldots,\ell+2}\ C[h_3]\ \mathcal{E}\ ;\ x_{\ell+3} \div\!\leftarrow x_0\ ;$

$\bullet$

$\bullet$

$\bullet$

$\bullet$

$\mathcal{B}_{1,2,\ldots,\ell+j-2}\ C[h_{j-1}]\ \mathcal{E}\ ;\ x_{\ell+j-1} \div\!\leftarrow x_0\ ;$

$\mathcal{B}_{\ell+1,\ell+2,\ldots,\ell+j-1}\ C[h_j]\ \mathcal{E}\ ;\ x_j \div\!\leftarrow x_0\ ;$

$x_{j-1} \div\!\leftarrow x_{\ell+j-1}\quad ;\quad x_{j-2} \div\!\leftarrow x_{\ell+j-2}\quad ;\quad \bullet\ \bullet\ \bullet\quad ;\quad x_1 \div\!\leftarrow x_{\ell+1}\quad ;\quad C[f]\ )$

Finally, for minimization, we suppose given $C[f]$ strongly computing some function f of "k" variables, where $k > 1$, and wish to produce a command $C[g]$ for g defined by

$$g(v_1, \cdots, v_{k-1})\ :=\ \min\{\ w\ :\ f(v_1, \cdots, v_{k-1}, w) = 0\ \}\ ,$$

with domains as explained on p.403 of **[LM]**. (See also the subsection **Minimization** in Section V below.) Here it is :

$(x_k \div\!\leftarrow 0\ ;$
$\mathcal{B}_{1,2,\ldots,k}\ C[f]\ \mathcal{E}\ ;$
while $0 < x_0$
do $\quad(x_k \div\!\leftarrow x_k + 1\ ;\ \mathcal{B}_{1,2,\ldots,k}\ \ C[f]\ \ \mathcal{E}\quad)\ ;$
$x_0 \div\!\leftarrow x_k\ )$

This finishes the proof that $\mathcal{RC} \Rightarrow \mathcal{BC}$, rather easy compared to any proofs of either $\mathcal{RC} \Rightarrow \mathcal{LC}$ or $\mathcal{RC} \Rightarrow \mathcal{TC}$. The 'machine languages' for λ-definability and Turing computability are awkward compared to the 'higher level language' **BTEN**, even though the latter is very weak *in the practical sense* compared to the real (usable) computer languages which are its parents. It is certainly as strong *theoretically*, even without the **ite**-command ! See Subsection IV-2.

The **BTEN** programs in this section above are clearly 'correct'. But in case you'd like to see a kind of *formal proof* of this sort of claim, look ahead to Section VIII and its references, where we discuss formal logic for program correctness.

The proof in the next section of the converse, $\mathcal{BC} \Rightarrow \mathcal{RC}$, is necessarily much longer, but really based on a single idea, namely that of coding commands by numbers. Actually, if you assume Church's thesis, then $\mathcal{BC} \Rightarrow \mathcal{RC}$ is immediate, since clearly no reasonable person can deny that Babbage computability really does mean that there is an effective mechanical process to compute the function. It happens all the time, right there on your desktop! The $\mathcal{RC} \Rightarrow \mathcal{BC}$ half we just did, showing that **BTEN** is theoretically at least as strong as any possible command language, is the important half. But we should give a proof of the other half, since later, we are going to prove the Gödel express in the form $\mathcal{RC} \Rightarrow$ *expressible* , whereas, after that, we'll check that the 'algorithms' in the proofs of the Church, Gödel and Tarski theorems really are algorithms, by writing **BTEN** programs for them.

In fact, the next section contains some material much more basic than merely a "proof . . . of the converse". It introduces one of the most pervasive ideas of all, namely, *Gödel numbering*.

At this point sometime soon, there will be a discussion of using $1^{\underline{\text{st}}}$order languages other than number theory as a basis for any of **ATEN**, **BTEN**,'while' (that is, **ATEN** supplemented by **if-then-else**-commands, or equivalently, **BTEN** with $\mathcal{B} \ldots \mathcal{E}$ removed) and also for more complicated command languages closer to ALGOL. Then some exercises will be given, in the spirit of Subsection IV-2 ahead, showing how any recursive function can be computed by $\mathbf{ATEN}_{Peano,\mathbf{N}}$, a much more austere language than $\mathbf{BTEN}_{numbertheory,\mathbf{N}}$ used above. By using $\mathbf{ATEN}_{Peano,\mathbf{N}}$ we simplify the command language's structural inductive definition from five clauses to three, stick with the natu-

ral numbers as a set, but use a considerably simplified $1^{\underline{\text{st}}}$order language, so the structure on $\mathbf{N}$ just has the successor function and the constant $0_{\mathbf{N}}$.

Then the proof in the next section can be made shorter and more palatable, since the number of things to code and cases to consider is very much reduced by using the simpler language. Pedagogically, this is somewhat attractive as well, providing some exercises showing a wide variety of command languages, each of which is quite sufficient as a definition of *computability*.

IV: Proof that $\mathcal{BC} \Rightarrow \mathcal{RC}$; Kleene's computation relation; the universal command and the halting problem.

To completely master all the details in this section, Appendix A (on recursive functions and on basic coding) must be absorbed. But the reader might prefer to first read the earlier parts of this section, particularly the following subsection. (If you happen to be allergic to performing a suspension of disbelief, then read this section in the order IV-2, then IV-3, then IV-1, of its subsections, and delay Subsection IV-3 until after reading Appendix A, a hopefully logically impeccable development. After that, Subsections IV-4 to IV-12 are extras, a short course in recursion theory, not needed for the purpose of making rigorous the proofs at the end of [**LM**].)

IV-1. Big Picture

This should help to motivate wading into the thicket of technical details in Appendix A and in the other subsections here. The general point to be made is about a pattern of argument, due mainly to Kleene and Turing, which is rather ubiquitous in this context. Books on logic or recursion theory will often introduce two possible definitions of 'computable function' :

(1) a definition of *recursive* (our $\mathcal{RC}$) where there is little latitude in wording the definition; and

(2) a more 'physical' or 'mechanical' definition : our $\mathcal{BC}$, but much more often, Turing machines ($\mathcal{TC}$) (see [**M**] for a slightly different version than ours, and a take-no-prisoners-style proof that $\mathcal{TC} \Rightarrow \mathcal{RC}$)), or register machines (see [**H**]), or "the basic machine" (see [**S**]), or "the abacus machine" (see [**B&J**] Ch. 6), etc. The latter four are more like 'hardware definitions' than our 'software definition'. Actually the register machine versions, especially the elegant one in the CS-style text [**D-S-W**], are as much software as hardware, but more like FORTRAN than PASCAL.

Whichever setup from (2) is used, the Kleene approach introduces a function $KLN(c, \vec{v}, h)$ of three variables that is a relation—i.e. total, and taking the values 1 ("true") and 0 ("false"). The variable c is a natural number which codes the command (or the program, or the particular machine, etc., depending on the setup choice from (2) above). The variable $\vec{v}$ is a vector $(v_1, \cdots, v_t)$ of natural numbers which represents the input. So actually $KLN = KLN_t$ is a relation on "t+2" natural number variables, and there is a

different KLN relation for every $t \geq 1$. Finally, the variable h is to be a natural number which [at least when the relation is 'true', i.e. $KLN_t(c, \vec{v}, h) = 1$], is supposed to code the 'history' of the computation, the details concerning the finite number of 'steps' which the machine takes.

Thus there will be essentially only one h for which $KLN_t(c, \vec{v}, h) = 1$ in the situation where, if command $\#c$ (or program $\#c$, or machine $\#c$) is activated with $\vec{v}$ as the input [i.e. $(0, \vec{v}, \underline{0})$ as the initial bin configuration], then in fact it turns out that the computation eventually terminates. In all other situations, we define $KLN_t(c, \vec{v}, h) = 0$, these situations being when c is not the code number of any command, or h is not the code number of any history, or the wrong history, or when computation $\#c$ with input $\vec{v}$ results in an infinite loop.

This Kleene computation relation will be easier to work with mathematically, but, because of h's essential uniqueness above, lurking in the background is the important **partial** function

$$(c, \vec{v}) \mapsto \begin{cases} h & \text{when } KLN_t(c, \vec{v}, h) = 1 \ ; \\ \text{undefined} & \text{when } KLN_t(c, \vec{v}, h) = 0 \ . \end{cases}$$

This maps a (command plus input) to the computational history which results. It surely seems a very natural thing to look at!

Now the lengthy but elementary proof in Subsection IV-3 below is for the following.

Theorem IV-1.1: *The relation KLN_t is a recursive function* (actually even *primitive* recursive, as defined in Appendix A).

The partial function in the display just above is evidently given by

$$(c, \vec{v}) \mapsto \min\{ \ h \ : \ \overline{KLN}_t(c, \vec{v}, h) = 0 \ \} \ ,$$

where $\overline{KLN}$ is KLN followed by switching 0 and 1 with each other. A corollary of the theorem is that this partial function is also recursive, being a minimization of something recursive. [Note that, by minimizing here, we've eradicated the slight ambiguity in choice of h, that ambiguity explained below.]

Here is some more information on what will be meant by 'history' in our setup, that is, $\mathcal{BC}$, a program then being a **BTEN**-command. Our

mechanical picture of what actually occurs in the process of computation tells us the following :
(1) As the (eventually terminating) computation proceeds, one sees a finite sequence of changes in the contents of the bins.
(2) Each change is 'caused' by a particular **atomic sub-command** (that is, a connected substring of the form $x_j \div s$) of the overall command (or 'program'). We are here ignoring the $\mathcal{B}$—$\mathcal{E}$ style of **BTEN**-command construction, as explained in the next subsection.
(3) Which one is the next atomic sub-command to be executed depends on which one we are at now and what the bin contents are after executing this one.

The (uncoded) history of a (terminating) computation will then be a tuple with an odd number of slots. Starting with the first slot, namely the $(\ell+1)$-tuple $(0, v_1, \cdots, v_t, 0, \cdots, 0)$ which is the input, the 3rd, 5th, etc. slots will also be $(\ell+1)$-tuples $(w_0, w_1, \cdots, w_\ell)$ of natural numbers, namely, the successive bin contents. Here we take ℓ to be at least as large as the maximum j for which the variable x_j occurs in the program; and also ℓ must be at least as large as t. [This non-uniqueness of ℓ is the "slight ambiguity" referred to just above. It is simpler to leave it in. That is, there will be lots of h with $KLN_t(c, \vec{v}, h) = 1$ if there are any, but all these different h are related very simply.] All other bin contents, in bin i for $i > \ell$, will of course be 0 and remain being 0.

Finally, the 2nd, 4th, etc. slots in the (uncoded) history will be *marked commands*. Ignoring the marking, all these will be the same, namely that command coded by c. The marking will amount to specifying which atomic sub-command is executed in creating the jump from the $(\ell+1)$-tuple immediately to its left to the one immediately to its right. [You might perhaps picture a marked command as the string of symbols which is the underlying command (or program) with the *mark*, for example a circle, or perhaps rectangle, as we do in places below, *surrounding* one particular connected substring of the form $x_j \div s$.]

The number of 'steps' in the computation is just less (by 1/2) than half the number of slots in its history. Each slot is determined by the two immediately to its left, except for the first slot, which is really the input, and the second, which can depend on the first, for example in the case of a **whdo** command. (See the next subsection concerning the possibility of ignoring **ite** commands.)

Finally, the information which gives the total number of slots in the history, and signals termination of the computation, thereby determining the number of 'steps', is that the penultimate slot in the history has as its marked atomic sub-command one 'close enough to' the right-hand end of the command, and the last slot is a collection of bin contents which will fail any '**whdo** test' which that last marked atomic sub-command might be 'inside', and also any '**whdo** test' which comes after that. And so there is no atomic sub-command to move to at this point, and the computation terminates.

The above is rather 'loosey-goosey', to say the least. But the reader may rest assured that conversion to mathematically precise form won't be terribly difficult, just mildly tedious. Basically, this will amount to doing the requisite encoding. The above few paragraphs are simply our mental picture of how a **BTEN** command is converted into a mechanical process of successively altering the bin contents in our machine.

In addition to the partial recursive function above minimizing the Kleene computation relation, we need a total recursive function $PRINT$ which picks out the computed function value. This will be more-or-less trivial in any setup. In ours (namely $\mathcal{BC}$) starting from the code number h of the history, it just has to pick out the last slot in the history (the final bin contents), and then pick out the first slot (that is, the contents of bin 0) in the $(\ell+1)$-tuple which is in that last slot of the history. Actually, $PRINT$ will turn out to also be primitive recursive.

And now there is nothing much to the

Proof that $\mathcal{BC} \Rightarrow \mathcal{RC}$:

If $D_0 \subset \mathbf{N}^t$, and $f_0 : D_0 \to \mathbf{N}^t$ is a $\mathcal{BC}$-function, let c_0 be the code of some command which computes f_0. Then, for all $\vec{v} \in D_0$, we have

$$f_0(\vec{v}) \;=\; PRINT(\min\{h : KLN_t(c_0, \vec{v}, h) = 1\}) \;.$$

Thus f_0 is the composite of two recursive functions, namely $PRINT$ and

$$\vec{v} \;\mapsto\; \begin{cases} \min\{h : \overline{KLN}_t(c_0, \vec{v}, h) = 0\} & \text{when } \vec{v} \in D_0 \;; \\ \text{undefined} & \text{when } \vec{v} \notin D_0 \;. \end{cases}$$

Thus f_0 is recursive, i.e. $\mathcal{RC}$, as required.

So now we know (modulo proving in Subsection IV-3 that the Kleene relation is recursive) that $\mathcal{BC} \iff \mathcal{RC}$: being computable by a **BTEN** command is equivalent to being recursive. Thus, the above also proves one of Kleene's big theorems :

Theorem IV-1.2: (Kleene Normal Form): *Any recursive function can be written as the composite of two functions G and M, where G is a primitive recursive function of one variable, and M is the minimization of a primitive recursive function H.*

The functions G and H are total, of course.

Next in this big picture is to discuss **universal** machines/programs/commands. As mentioned before, the universal *Turing machine* is the standard theoretical CS model for an actual computer. But here we're doing $\mathcal{BC}$, not $\mathcal{TC}$, so will phrase things in that setup.

Imagine allowing c to vary in the previous proof, producing the recursive function U_t as follows :

$$U_t : (c, \vec{v}) \mapsto \begin{cases} \text{undefined} & \text{if } \forall h \ KLN_t(c, \vec{v}, h) \neq 1; \\ PRINT(\min\{h : KLN_t(c, \vec{v}, h) = 1\}) & \text{otherwise.} \end{cases}$$

The partial function U_t has domain

$$D_t \ = \ \{ \ (c, \vec{v}) \ : \ \exists h \in \mathbf{N} \text{ with } \ KLN_t(c, \vec{v}, h) = 1 \ \} \ \subset \ \mathbf{N}^{t+1} \ .$$

So D_t consists of pairs (code number c of command C, t-tuple input $\vec{v}$) for which carrying out command C with bins initialized as $(0, \vec{v}, \underline{0})$ will result in a terminating computation; that is, in the notation of Section II,

$$||C||(0, \vec{v}, \underline{0}) \ \neq \ err \ .$$

By the previous proof, *every* computable function of t variables is computed by a command which computes U_t, merely by inserting the extra number into its input which is the code number of any choice of command that computes the function.

A more mathematical version is this :

Theorem IV-1.3: *For each $t \geq 1$, there is a recursive function U_t of '$t+1$' variables such that, for every recursive function f of 't' variables, there exists a number c_f so that, for every $\vec{v} \in \mathbf{N}^t$,*

$$\vec{v} \in \text{domain}(f) \iff (c_f, \vec{v}) \in \text{domain}(U_t) \ ,$$

and, for such $\vec{v}$,

$$f(\vec{v}) \ = \ U_t(c_f, \vec{v}) \ .$$

There is also a sense in which U_1 is universal for *all* recursive functions, that is, functions of *any* number of variables. For each $t > 0$, fix a recursive bijection (CAN for Cantor—see Appendix A for details)

$$CAN_t \ : \ \mathbf{N}^t \ \to \ \mathbf{N} \ .$$

Its inverse $\mathbf{N} \ \to \ \mathbf{N}^t$ is given by

$$d \ \mapsto \ (P_{1,t}(d), P_{2,t}(d), \cdots, P_{t,t}(d) \ ,$$

where the component functions $P_{i,t} : \mathbf{N} \to \mathbf{N}$ are also total recursive functions. With notation as in the theorem, we have

$$f(\vec{v}) \ = \ f(P_{1,t}(CAN_t(\vec{v}), P_{2,t}(CAN_t(\vec{v}), \cdots, P_{t,t}(CAN_t(\vec{v})) \ .$$

Consider the recursive function g_f of one variable given by

$$g_f(n) \ := \ f(P_{1,t}(n), P_{2,t}(n), \cdots, P_{t,t}(n)) \ ,$$

so that f is the composite of g_f with CAN_t . By the previous theorem, choose a c_{g_f} such that $g_f(n) = U_1(c_{g_f}, n)$ for all n in the domain of g_f. Thus

$$f(\vec{v}) \ = \ g_f(CAN_t(\vec{v})) \ = \ U_1(c_{g_f}, CAN_t(\vec{v}))$$

for all $\vec{v}$ in the domain of f.

Now, after carefully checking domains, we have proved the following.

Theorem IV-1.4: *There is a recursive function U* (namely U_1 previous) *of two variables such that, for every $t \geq 1$, and every recursive function f of 't' variables, there exists a number d_f* (namely c_{g_f} previous) *so that, for every $\vec{v} \in \mathbf{N}^t$,*

$$\vec{v} \in \text{domain}(f) \iff (d_f, CAN_t(\vec{v})) \in \text{domain}(U) \ ,$$

and, for all such $\vec{v}$,

$$f(\vec{v}) \ = \ U(d_f, CAN_t(\vec{v})) \ .$$

Thus U is a single function of two variables such that a fixed program (i.e. **BTEN** command) which computes U will compute *every* computable function f of any number of variables, as follows:

First use an easy fixed program for computing CAN_t to convert the input tuple into a single number.

Then supplement this by a second single number which codes (a command for) f (the 'compiler').

Finally, the fixed universal program for U is applied to that pair.

That computation will terminate if and only if the original input tuple is in the domain of f, in which case bin 0 will then contain the correct answer.

(The word "easy" above refers to the fact that the program will likely be much simpler than the universal program, and also very much shorter for t of reasonable size.)

> *Thus, from the point of view of recursive-function theory, there is no essential distinction between singulary and n-ary ($n > 1$) functions. It is as though, in analysis, one possessed analytic homeomorphisms between one-dimensional and n-dimensional ($n \geq 1$) euclidean space.*
>
> Martin Davis

If we had studied $\mathcal{TC}$ instead of $\mathcal{BC}$, everything so far in this section would have been identical, except for the rough details about the history of a computation. Then any Turing machine for computing a function U with the properties in the last theorem is called a **universal Turing machine**.

The legal and academic disputes about just exactly who invented the computer, Turing/Von Neumann or the engineers, seem vastly amusing in retrospect. In the distant future, when we have the pleasure of surveying a few million alien civilizations about their history of inventing their computers, that invention will have occurred at different times, both before and after the invention of vacuum tubes, transistors, etc. But always **after** at least intuiting the above theorems (to account for Mr. Babbage!), and likely after actually proving them, since the ideas in the proof are not unrelated to the construction plan for a genuine computer. These remarks are undoubtedly out-of-line, coming from a non-expert. Rebuttals from any source would be more than welcome!

Consideration of the halting problem and related material will now complete our big picture here.

Inspired by Cantor's diagonal method for proving the uncountability of the reals, we can get a couple of important **non**-computable functions by the following method.

Consider the array below of the 'values' of the universal function U just above (we include *err* as the value of $U(i,j)$ when (i,j) is not in the domain of U):

$$\begin{array}{cccc} U(0,0) & U(0,1) & U(0,2) & U(0,3) \\ U(1,0) & U(1,1) & U(1,2) & U(1,3) \\ U(2,0) & U(2,1) & U(2,2) & U(2,3) \\ U(3,0) & U(3,1) & U(3,2) & U(3,3) \\ \bullet & \bullet & & \bullet \\ \bullet & \bullet & & \end{array} \quad \begin{array}{ccc} \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet \\ & & \\ & & \bullet \end{array}$$

A 1-variable function $f : D \to \mathbf{N}$, where $D \subset \mathbf{N}$, may be identified with its sequence of values, $f(0)\ \ f(1)\ \ f(2)\ \ \bullet\ \ \bullet$, where again we include *err* as $f(i)$ for those $i \notin D$. By **IV-1.3** with $t = 1$, any computable f appears at least once (in fact many times) as a row in the array above.

So now imagine yourself moving step-by-step down the main diagonal in the above array, and choosing a sequence whose nth term dis<u></u>agrees with $U(n,n)$, for every n. Then such a sequence corresponds to a function which is *not* computable, since it cannot be the nth row in the array for any n (because the nth entries disagree).

Our first specific way to do this produces a non-computability not quite so important as the second one further down. In this case, always replace $U(n,n)$ by 0, except if $U(n,n) = 0$, in which case, we replace it by 1. The

corresponding function R is total, taking only values 0 and 1. So we often would think of it as a *relation*, where *true* corresponds to 1, and *false* to 0.

What is this relation R? Clearly we have

$$R(n) \;=\; \min\{\; b \;:\; U(n,n) \neq b \;\}\;.$$

So R holds only for those n with $U(n,n) = 0$.

Now, to see some 'significance' in this, let's go back to Kleene's computation relation $KLN = KLN_1$:

$$KLN(c,v,h) = \begin{cases} 1 & \text{when command\#}c \text{ with input } v \text{ produces history\#}h\ ; \\ 0 & \text{otherwise}\quad . \end{cases}$$

Its (primitive) recursiveness is fundamental to the theory here. But why go to the trouble of fussing with the entire history of a computation? Why not just consider the following relation K' ? Define $K' : \mathbf{N}^3 \to \mathbf{N}$ by

$$K'(c,v,b) = \begin{cases} 1 & \text{when command\#}c \text{ with input } v \text{ produces output } b\ ; \\ 0 & \text{otherwise}\quad . \end{cases}$$

It appears that this function will do everything for us which KLN did, and in a simpler way. But wait! Is K' recursive? The answer is **no**. Since $U(n,n) \neq b \iff K'(n,n,b) \neq 1$, we see that

$$R(n) \;=\; \min\{\; b \;:\; K'(n,n,b) = 0 \;\}\;.$$

So, if K' were recursive, so would be $(n,b) \mapsto K'(n,n,b)$. But then so would R, which we know *isn't* computable.

And so K' cannot be used in place of Kleene's relation. If it had been primitive recursive, by inspecting its definition, we would quickly see that any recursive function would be writeable as the minimization of a primitive recursive function—and that's too strong an improvement on Kleene's normal form to be true. As an exercise here, find a specific recursive function which is not the minimization of any primitive recursive function.

A little intuitive thinking convinces one that our contrast between KLN and K' makes sense: Given a command and an input, there is an intuitively obvious mechanical procedure for deciding whether or not a proposed computational history **is** actually what happens when the command is run with that input. But there is no obvious mechanical procedure for deciding whether

or not a proposed output is the result when the command is run with that input. There's an algorithm which will terminate with the answer "yes" if the proposed output is correct; but if incorrect, the algorithm I'm thinking about sometimes fails to terminate, namely when the command itself fails to terminate on the given input. This latter algorithm corresponds to the fact that, despite R's non-recursiveness, the partial function

$$n \;\mapsto\; \begin{cases} 1 & \text{if } R(n) = 1 \ ; \\ err & \text{if } R(n) = 0 \ . \end{cases}$$

is computable. See the first example just after **IV-5.4**.

Our second example is simply to replace, in the diagonal of the array further up, all $U(n,n)$ which are defined (i.e. numbers) by err ; and replace all copies of err by the number 0. This gives a partial function of one variable which, by construction, is non-computable. As we see below, this is the main step in Turing's proof that the 'halting function' is non-computable, where the function just defined is called E. Because of its fundamental importance, we'll be more formal just below about this theorem.

By composing with the constant function mapping all of $\mathbf{N}$ to the number 1, we see from the first partial recursive function considered in this section that

$$(c, \vec{v}) \;\mapsto\; \begin{cases} 1 & \text{if } KLN_t(c, \vec{v}, h) = 1 \text{ for some } h \ ; \\ \text{undefined} & \text{otherwise} \ ; \end{cases}$$

is a perfectly fine **partial** recursive function.

But contrast that with :

Theorem IV-1.5: (Turing's unsolvability of the halting problem) *The total function,* $HAL_t : \mathbf{N}^{t+1} \to \mathbf{N}$, *defined by*

$$HAL_t \;:\; (c, \vec{v}) \;\mapsto\; \begin{cases} 1 & \text{if } KLN_t(c, \vec{v}, h) = 1 \text{ for some } h \ ; \\ 0 & \text{otherwise} \ , \end{cases}$$

is **not** *a recursive function.*

This is the function which would decide for us, if only we could compute it (which we can't!) whether a given program on a given input will halt

or not. That is, will it give a terminating computation? Thus, there is no algorithm which, given a **BTEN** command C and a proposed input $\vec{v}$, will decide whether or not $||C||(0, \vec{v}, \underline{0}) \neq err$. That is, to say it again Sam, you can't detect the possibility of "infinite loops" coming up, beforehand, by a single bug-detecting program. This is related to the early discussion here, Cantor's Curse on the Constructivists. The theorem and its proof are a mathematical version of the statement towards the end of that discussion about the non-existence of an algorithm for deciding membership in domains of computable functions from algorithms to compute those functions.

On the other hand, the closely related **non-total** recursive function just before the theorem statement, together with our early discussion of "algorithmic listing" vis-a-vis "not-always-terminating-algorithms" (see also Subsection IV-5), shows that there is a mechanical process which lists all pairs $(c, \vec{v})$ for which c is the code of a program which halts when applied to input $\vec{v}$. To re-iterate for the 5th time the previous paragraph, this latter true assertion is strictly *weaker* than falsely asserting that there is a mechanical process which, for every $(c, \vec{v})$, will decide whether it comes up as a member of the list ! It is clear from the informal, intuitive point of view that such a mechanically generated list must exist.

The above will hopefully remind you of Church's Theorem giving a negative solution Hilbert's Entscheidungsproblem. That is, despite the fact that there is a mechanical process which lists all formulae deducible from a given decidable set of premisses in $1^{\underline{\text{st}}}$order number theory (for example), there is no mechanical process which, given the formula beforehand, will decide whether it is deducible. Turing solved that problem also, independently of Church, using precisely the theorem above. One of the chief aims we have here is filling the gap in the proof in [**LM**] of Church's theorem. This is done in Subsection VI-4.

Proof of Turing's theorem when $t = 1$:
We shall prove the stronger result that the one variable function

$$n \mapsto HAL_1(n, n)$$

is not recursive. Assume for a contradiction that it *is* recursive. Let f be the composite

$$f \ : \ \mathbf{N}^2 \ \to \ \mathbf{N} \ \to \ \mathbf{N}$$
$$(c,n) \mapsto \ n \mapsto \ HAL_1(n,n)$$

Then f is a total recursive function, being a composite of such functions. Now define

$$E(n) \ := \ \min\{ \ c \ : \ f(c,n) = 0 \ \} \ .$$

Then

$$E(n) \ = \ \begin{cases} 0 & \text{if } HAL_1(n,n) = 0 \ ; \\ \text{undefined} & \text{if } HAL_1(n,n) = 1 \ . \end{cases}$$

By its definition in the penultimate display, E is a partial recursive function. So let c_0 be the code of a **BTEN**-command which computes it. Then we get our desired contradiction as follows :

$$HAL_1(c_0,c_0) = 0 \iff E(c_0) = 0 \iff E(c_0) \text{ is defined } \iff HAL_1(c_0,c_0) = 1 \ .$$

The former two 'iffs' are immediate from the display just above. The last one holds because

$$\forall n \ [HAL_1(c_0,n) = 1 \Leftrightarrow [KLN_1(c_0,n,h) = 1 \text{ for some } h] \Leftrightarrow n \in \text{domain}(E)] \ ;$$

the left-hand 'iff' comes directly from the definition of HAL_1; and the right-hand one because c_0 is the code of a command for computing E .

In 1936, the notion of a computable function was clarified by Turing, and he showed the existence of universal computers that, with an appropriate program, could compute anything computed by any other computer. All our stored program computers, when provided with unlimited auxiliary storage, are universal in Turing's sense. In some subconscious sense, even the sales departments of computer manufacturers are aware of this, and they do not advertise magic instructions that cannot be simulated on competitors' machines, but only that their machines are faster, cheaper, have more memory, or are easier to program.

The second major result was the existence of classes of unsolvable problems. This keeps all but the most ignorant of us out of certain Quixotic enterprises, such as trying to invent a debugging procedure that can infallibly tell if a program being examined will get into a loop.

John McCarthy

This major discovery of Turing, as stated in the theorem, is independent of the (yet to be proved) fact that KLN is recursive. We used the function, but not its recursivity; that $\mathcal{RC} \Rightarrow \mathcal{BC}$ was used, but not its converse. However the remarks after the theorem statement, about the non-existence of a **BTEN** algorithm, do depend on knowing that $\mathcal{BC}$ and $\mathcal{RC}$ are the same, and that depends on recursivity of KLN. And, in any case, the 'meaning' of c does depend at least on having encoded the language **BTEN**.

The proof of this theorem when t is general will be left as an exercise. It uses the 'same' argument as above, combining it with the recursive bijection $CAN_t : \mathbf{N}^t \to \mathbf{N}$. The "busy beaver function", earlier proved non-computable, can also be used to prove Turing's theorem.

As mentioned, the halting problem relates to our early discussion of the impossibility of building a self-contained cogent theory of strictly *total* computable functions—see Cantor's Curse on the Constructivists, page 7. The following two theorems are more exact mathematical analogues of the main informal argument given there. We'll just consider functions of one variable and leave the general case as an exercise (both formulation and proof).

Theorem IV-1.6:(weaker) *The relation* $TOT' : \mathbf{N} \to \mathbf{N}$ *given by*

$$c \mapsto \begin{cases} 1 & \text{if } c \text{ is the code of a command with } \forall n \exists h \ KLN_1(c,n,h) = 1 \ ; \\ 0 & \text{otherwise .} \end{cases}$$

is not recursive.

In other words, there is no algorithm which decides whether the function of one variable which a given **BTEN**-command computes is total or non-total.

Theorem IV-1.7: (stronger) *The partial function* TOT *of one variable given by*

$$c \mapsto \begin{cases} 1 & \text{if } c \text{ is the code of a command with } \forall n \exists h \ KLN_1(c,n,h) = 1 \ ; \\ err & \text{otherwise .} \end{cases}$$

is not recursive.

In other words, there is not even an algorithm which *lists* all the (codes of) **BTEN**-commands which compute **total** functions of one variable. We get this from the early informal discussion on listing algorithms and non-termination of ordinary algorithms, or its more rigorous version discussing recursive enumerability (in Subsection IV-5).

It is clear from the statements after them that the stronger theorem does imply the weaker one. More mathematically, it is easy to prove that [recursiveness of TOT'] $\Rightarrow$ [recursiveness of TOT] :

$$TOT(d) \;=\; 1 + \min\{\; c \;:\; H(c,d) = 1 \;\}\;,$$

where $H(c,d) := TOT'(d)$; i.e. H is a projection followed by TOT' .

Proof of the stronger theorem. For a contradiction, assume that TOT is $\mathcal{RC}$. Since there are infinitely many total recursive functions of one variable, there are arbitrarily large numbers c with $TOT(c) = 1$, so the function M given by

$$M(b) \;:=\; \min\{\; c \;:\; c \geq b \;\text{ and }\; TOT(c) = 1 \;\}$$

is total recursive. Since $TOT(M(b)) = 1$, by the definition of TOT it follows that for all (a,b), there is an h with $KLN_1(M(b),a,h) = 1$. So we get a total recursive function L of two variables with the definition

$$L(a,b) \;:=\; PRINT(\min\{\; h \;:\; KLN_1(M(b),a,h) = 1 \;\}\;.$$

Now 'diagonalize out' by defining $D : \mathbf{N} \to \mathbf{N}$ by $D(b) := 1 + L(b,b)$, clearly a total recursive function. Thus D may be computed by a **BTEN**-command whose code is d, say. Since D is total, the definition of TOT gives $TOT(d) = 1$. But then the definition of M gives $M(d) = d$.

By the definition of d, for every n the unique h with $KLN_1(d,n,h) = 1$ satisfies $PRINT(h) = D(n)$, using the definitions of KLN_1 and $PRINT$. Thus, by the definition of L,

$$\forall n \qquad L(n,d) \;=\; D(n) \qquad\qquad (*)$$

But now we get a contradiction :

$$D(d) \;=\; 1 + L(d,d) \;=\; 1 + D(d)\;,$$

the first equality by the definition of D, and the second by $(*)$.

Slightly hidden within this proof by contradiction is a purported effective listing L of all the total recursive functions of one variable. Then D is the total recursive function that cannot be in the list, by 'diagonalizing out', giving the contradiction. This proof is the formal counterpart of our much earlier informal argument concerning the futility of trying for a self-contained theory of **total** computable functions.

Note again that all but one of the major results in this subsection are dependent on proving the basic fact that KLN_t **is** recursive (and also $PRINT$, but that one will be easy). We shall continue asking you for a suspension of disbelief in this for one more subsection, before wading into its proof in Subsection IV-3. However, it is quite clear intuitively from its definition that KLN_t is computable in the sense that we can concoct an informal algorithm for it. So Church's thesis makes it clear that it must be recursive.

To finish this section, here are a few comments contrasting our "Babbage computability" with a typical "register machine" approach (as in **[H]** for example). This depends on the encodings ahead in Appendix A. It is peripheral to the main purpose here, so we have banished it to small print.

Our picture of a **BTEN** computation is based on a sequence of registers, or bins, as we call them, so there is considerable similarity. It is probably true that the encoding work here, needed to prove that KLN is recursive (see Subsection IV-3), is somewhat more tedious than that for the more 'FORTRAN-like' register machine instructions, which are numbered, so that GOTO instructions can be used. On the other hand, compare the brief proof below to the corresponding one (most of pp. 410 and 411) in **[H]**.

One Parameter Theorem IV-1.8: *For any "$\ell+1$"-ary recursive function f, there is a recursive $g : \mathbf{N} \to \mathbf{N}$ such that, for all $(\vec{v}, b)$, we have $f(\vec{v}, b) = U_\ell(g(b), \vec{v})$.* [This says that commands C_b can be chosen which compute $\vec{v} \mapsto f(\vec{v}, b)$ in such a way that $b \mapsto \#C_b$ is recursive. The equation in the theorem is intended to include the statement that

$$(\vec{v}, b) \in \text{domain}(f) \Longleftrightarrow (g(b), \vec{v}) \in \text{domain}(U_\ell) \ .]$$

Proof. Let $C[f]$ compute f, and define $g(b) :=$ code of $(x_{\ell+1} \Leftarrow$ "b" ; $C[f])$. The latter command obviously computes $\vec{v} \mapsto f(\vec{v}, b)$, so $f(\vec{v}, b) = U_\ell(g(b), \vec{v})$, by the definition of U_ℓ. Now one can simply write down explicitly that code above to see that g is recursive. Merely saying that should suffice. But if you insist, here is $g(b)$ in gory detail, using Appendix A and Subsection IV-3 encodings :

$$< 2, \#(x_{\ell+1} \Leftarrow \text{"}b\text{"}), \#C[f] > \ = \ < 2, < 1, < 0, \ell, \ell+1 >, SYN(b) >, \#C[f] > \ .$$

A statement of this which is stronger in two respects, but refers only to universal functions, is the following.

At Least One Other Parameter Theorem IV-1.9: *For any ℓ, n, there is a primitive recursive function $V_{\ell,n}$ of "$n+1$" variables, such that the universal recursive functions U_i are related by*

$$U_{\ell+n}(c, b_1, \cdots, b_n, a_1, \cdots, a_\ell) = U_\ell(V_{\ell,n}(c, b_1, \cdots, b_n), a_1, \cdots, a_\ell)$$

See **[D-S-W]**, pages 87-88 for a proof involving a register machine setup.

For $n = 1$, we can just use the same proof as before, inserting c instead of $\#C[f]$. Then one can proceed by induction on n, iterating as in **[D-S-W]**.

But really, here it's simpler to use the same proof, just changing

$$g(b) := \text{code of } (x_{\ell+1} \leftarrow \text{“}b\text{”} \ ; \ C[f]) \qquad \text{to}$$

$$V_{\ell,n}(c, b_1, \cdots, b_n) := \text{code of } (x_{\ell+1} \leftarrow \text{“}b_1\text{”} \ ; \ \cdots \ ; \ x_{\ell+n} \leftarrow \text{“}b_n\text{”} \ ; \ \text{command}\#c) \ .$$

where the right-hand side really means

$$< \ 2 \ , \ \text{code of } (x_{\ell+1} \leftarrow \text{“}b_1\text{”} \ ; \ \cdots \ ; \ x_{\ell+n} \leftarrow \text{“}b_n\text{”} \ ) \ , \ c \ > \ .$$

And obviously the latter is a primitive recursive function.

IV-2. Economizing the language BTEN

Let **ATEN** be the **a**ustere cousin of **BTEN** in which we :
(1) eliminate the $\mathcal{B}_{i_1,\cdots,i_k}\cdots\mathcal{E}$ command construction;
(2) only allow $(C;D)$ in the formal command language, not $(C_1;C_2;\cdots:C_k)$ for $k>2$;
(3) and, perhaps surprisingly, eliminate the **ite** command construction.

Thus the syntax here is that an **ATEN**-command is any finite string of symbols generated by these three rules :

(i) $\boxed{x_i \dot{\leftarrow} t}$ is a command for each variable x_i and each term t in the assertion language.

(ii) $\boxed{(C;D)}$ is a command, for any commands C and D .

(iii) $\boxed{\mathbf{whdo}(F)(C)}$ is a command, for each (quantifier-free) formula F in the assertion language, and all commands C.

(N.B. The rectangles are for emphasis here, certainly not an integral part of the command, of course. This is mentioned not to insult your intelligence, but because rectangles as in (i) will be used as an integral part of the notation when we come to define marked commands in the next subsection!)

The main claim is that **ATEN**, despite appearances, is just as strong as **BTEN**. To sound pompous, we could say that **ATEN** is semantically equivalent to **BTEN**, despite being syntactically simpler. Of course, **ATEN** moves us even further away from practical programming languages such as C++ . And we'll want to have at least the richness of **BTEN** in Section VI below, to write out rigorous versions of the algorithms which appear informally in the proofs of the big logic theorems due to Gödel, Church, and Tarski from Appendix L in [**LM**].

On the other hand, in the next subsection, a certain degree of ugliness is avoided by only directly proving that $\mathcal{AC}\Rightarrow\mathcal{RC}$, ostensibly weaker than $\mathcal{BC}\Rightarrow\mathcal{RC}$. So here we wish to show how $\mathcal{BC}\Rightarrow\mathcal{RC}$ follows fairly easily. What is actually proved in the next section is that the Kleene computation relation is recursive, but the KLN we do is that for $\mathcal{AC}$-computability, defined using **ATEN**-commands, from which $\mathcal{AC}\Rightarrow\mathcal{RC}$ follows by the considerations of the previous subsection.

What exactly is meant by saying that **ATEN** is just as strong as **BTEN**? At minimum, it means that $\mathcal{BC} \Rightarrow \mathcal{AC}$ (the converse being obvious). That is, we want at least that any function computable by a **BTEN**-command is actually computable by an **ATEN**-command.

Perhaps the definition should be stronger: given any **BTEN**-command, we might want to require that there is an **ATEN**-command which terminates after starting with a given input if and only if the **BTEN**-command does, and does so with the *same* final bin values.

The last phrase however is stronger than anything needed later, and quite possibly false. For all purposes below, it is only necessary that the final bin contents from bin zero to bin "N" agree, for any pre-assigned N.

So what we are really claiming is the following equivalence between the smaller and larger command languages:

Given any $N \geq 0$ and any **BTEN**-command B, there is an **ATEN**-command A such that:

(i)$||A||(\underline{v}) = err$ if and only if $||B||(\underline{v}) = err$; and

(ii) if $||A||(\underline{v}) \neq err$, then $[||A||(\underline{v})]_i = [||B||(\underline{v})]_i$ for $0 \leq i \leq N$.

The technicalities needed to establish this follow below. There we solve the earlier exercise about the redundancy of the $\mathcal{B} \cdots \mathcal{E}$ construction, which is part of the above claim, as well as establishing the less 'obvious' redundancy of the **if-thendo-elsedo** construction.

The third economization is so trivial as to be barely worth mentioning : It is clear that, for example, $(C_1; (C_2; C_3))$ has the same effect as $(C_1; C_2; C_3)$ on bin values (and similarly for any other bracketing of any string like this of length ≥ 3) . We're saying that ";", as a binary operation, is *associative* up to equivalence. This is numbingly obvious, so let's move on to more interesting things and ignore this below.

Definition. Let C and D be **BTEN** commands, and let N be a natural number. Define $C \sim_N D$ to mean the following: given $\underline{v}$ and $\underline{w}$ such that $v_i = w_i$ for all $i \leq N$, we have :

$$||C||(\underline{v}) = err \iff ||D||(\underline{w}) = err$$

and

$$||C||(\underline{v}) \neq err \implies [||C||(\underline{v})]_i = [||D||(\underline{w})]_i \text{ for all } i \leq N \ .$$

This definition seems to be the correct notion. Roughly speaking it says that we don't care what is in bins beyond the Nth at the start of the calculation, and we care even less at the end, if any, of the calculation.

The definition doesn't quite give an equivalence relation, since reflexivity may fail. Here is an example. If C is the command $x_1 \leftarrow\!\!\div\ x_2$, then $C \sim_N C$ for all $N > 1$, but $C \not\sim_1 C$: consider for example $\underline{v} = (0, 0, 1, 0, 0, 0, \cdots)$ and $\underline{w} = (0, 0, 2, 0, 0, 0, \cdots)$. See Theorem **IV-2.2** below.

The following example, $C := \mathcal{B}_3(x_3 \leftarrow\!\!\div\ x_2\ ;\ x_2 \leftarrow\!\!\div\ x_1\ ;\ x_1 \leftarrow\!\!\div\ x_3)\mathcal{E}$, is a command whose effect is just to switch the contents of bins 1 and 2, and leave everything else as is. (Our example in Section II of a specific Turing machine did the same thing.) Now for $N \geq 3$, we define the command D_N in **ATEN** to be $(x_{N+1} \leftarrow\!\!\div\ x_2\ ;\ x_2 \leftarrow\!\!\div\ x_1\ ;\ x_1 \leftarrow\!\!\div\ x_{N+1})$. Clearly $C \sim_N D_N$. This command does mess up bin "$N + 1$". It illustrates the theorem below that given C and N, we can find a command which $\sim_N$'s C and has no $\mathcal{B}$ in it. But in this example, it is not clear at all whether we can find a single such command which $\sim_N$'s C for *all* sufficiently large N !

Prop. IV-2.1: *If $D \sim_N C$ for arbitrarily large N, then $||D|| = ||C||$.*

Proof. To show $||D|| = ||C||$, first note that $||D||(\underline{v}) = err$ if and only if $||C||(\underline{v}) = err$, taking $\underline{w} = \underline{v}$ in the definition. For any other $\underline{v}$, we must show $[||D||(\underline{v})]_i = [||C||(\underline{v})]_i$ for all i. Just choose N so that $D \sim_N C$ with N at least as large as i, again taking $\underline{w} = \underline{v}$ in the definition.

Theorem IV-2.2: *For any C, there is an ℓ such that $C \sim_N C$ for all $N \geq \ell$.*

Proof. Proceed by induction on commands C. It will be clear from the proof that we just need to take ℓ at least as large as all the subscripts on any variables and on any symbols $\mathcal{B}$ which appear in C.

When C is an atomic command, use the definition

$$||x_i \leftarrow\!\!\div\ t||(v_0, v_1, v_2, \cdots) := (v_0, v_1, \cdots, v_{i-1}, t^{\underline{v}}, v_{i+1}, \cdots) \quad \text{(see [LM], p.211)},$$

to motivate choosing ℓ to be the maximum of all the subscripts of variables in the term t. The result is immediate from the evident fact (see Lemma 6.2 in [**LM**]) that $t^{\underline{v}} = t^{\underline{w}}$ as long as $v_j = w_j$ for all j for which x_j appears in t.

When $C = (D; E)$ and we assume inductively that $D \sim_N D$ for all $N \geq j$

and $E \sim_N E$ for all $N \geq k$, take $\ell = \max(j,k)$. Then, for $N \geq \ell$, if $v_i = w_i$ for all $i \leq N$, we have, for $j \leq N$,

$$[||C||(\underline{v})]_j \;=\; [||E||(||D||(\underline{v}))]_j \;=\; [||E||(||D||(\underline{w}))]_j \;=\; [||C||(\underline{w})]_j \;,$$

The middle equality uses that $E \sim_N E$, and that $[||D||(\underline{v})]_i \;=\; [||D||(\underline{w})]_i$ for all $i \leq N$, since $D \sim_N D$. The other two equalities are just the definition of the semantics of the ";" construction. The calculation in the display is of course for when $||C||(\underline{v}) \neq err$. Otherwise, either $||D||(\underline{v}) = err$, or else $||D||(\underline{v}) \neq err$ and $||E||(||D||(\underline{v})) = err$. But the same two possibilities will now clearly hold with $\underline{v}$ replaced by $\underline{w}$.

When C is $\mathcal{B}_{i_1,\cdots,i_k} D \mathcal{E}$ and we assume inductively that $D \sim_N D$ for all $N \geq \ell$, again the result for C follows readily from the definition of the semantics of the $\mathcal{B} \cdots \mathcal{E}$ construction, for the same ℓ.

When $C =$**ite**$(F)(D)(E)$, and we assume inductively that $D \sim_N D$ for all $N \geq j$ and $E \sim_N E$ for all $N \geq k$, take ℓ to be the maximum of j, k and the subscripts of all variables appearing in the quantifier-free formula F. Then, for $N \geq \ell$, if $v_j = w_j$ for all $j \leq N$, we have, for $j \leq N$:

(i) If F is true at $\underline{v}$, then F is also true at $\underline{w}$, (truth depends only on the slots corresponding to variables actually appearing in the formula—see Theorem 6.1 in [**LM**]), and

$$[||C||(\underline{v})]_j \;=\; [||D||(\underline{v})]_j \;=\; [||D||(\underline{w})]_j \;=\; [||C||(\underline{w})]_j \;;\; \text{whereas}$$

(ii) if F is false at $\underline{v}$, then F is also false at $\underline{w}$, and

$$[||C||(\underline{v})]_j \;=\; [||E||(\underline{v})]_j \;=\; [||E||(\underline{w})]_j \;=\; [||C||(\underline{w})]_j \;.$$

The analysis of the case when $||C||(\underline{v}) = err$ is straightforward, as above.

Finally, when $C =$**whdo**$(F)(D)$, and we assume inductively that $D \sim_N D$ for all $N \geq j$, take ℓ to be the maximum of j and the subscripts of all variables appearing in the quantifier-free formula F. Then, for $N \geq \ell$, if $v_j = w_j$ for all $j \leq N$, we have the following for $j \leq N$. Proceeding by induction on k, with the cases $k = 0$ or 1 immediate by assumption, as long as $||D||^k(\underline{v}) \neq err$,

$$[||D||^k(\underline{v})]_j \;=\; [||D||(||D||^{k-1}(\underline{v}))]_j \;=\; [||D||(||D||^{k-1}(\underline{w}))]_j \;=\; [||D||^k(\underline{w})]_j \;.$$

In particular, the minimum k (if any) for which F is false at $||D||^k(\underline{v})$ is the same as the minimum with $\underline{v}$ replaced by $\underline{w}$. The required result is now straightforward, taking care with the analysis of when an infinite loop occurs:

Firstly, if $||C||(\underline{v}) = err$, then F is true at $||D||^k(\underline{v})$ either for all k, or until $||D||^k(\underline{v}) = err$. Therefore it is true at $||D||^k(\underline{w})$ either for all k, or until $||D||^k(\underline{w}) = err$, yielding $||D||(\underline{w}) = err$. The other way round is the same.

On the other hand, if $||C||(\underline{v}) \neq err$, then the minimums mentioned in the second previous paragraph agree, and the equality in the display, with k equal to that minimum, is exactly the required equality, by the definition of the semantics of the **whdo** construction.

Corollary IV-2.3: (Converse, and stronger, to Prop. 1.) *If* $||D|| = ||C||$, *then* $D \sim_N C$ *for all sufficiently large* N.

Proof. Choose some ℓ for C as in Theorem 2. Then for $N \geq \ell$, we get $D \sim_N C$ as follows. That $||C||(\underline{v}) = err \iff ||D||(\underline{w}) = err$ is clear. When $||C||(\underline{v}) \neq err$, proceed as follows. For any $j \leq N$, and any $\underline{v}$ and $\underline{w}$ such that $v_i = w_i$ for all $i \leq N$,

$$[||C||(\underline{v})]_j \;=\; [||C||(\underline{w})]_j \;=\; [||D||(\underline{w})]_j \ .$$

Theorem **IV-2.2** gives the first equality, and $||D|| = ||C||$ the second.

As indicated earlier, we won't expect to be able to replace a given C by a D from the smaller language in question with $D \sim_N C$ for all sufficiently large N. All we need is that, given C, and given sufficiently large N, we can find a D (which perhaps depends on N) for which $D \sim_N C$.

The next two propositions are very easy to prove. The first has been implicitly used a few times already, perhaps. The proofs will be left as a exercises.

Proposition IV-2.4: *For each* N, *the relation* $\sim_N$ *is symmetric and transitive* (but not reflexive!)

Proposition IV-2.5: *If* $C \sim_N D$ *for some* $N > 0$, *then the commands* C *and* D *compute the same function of "k" variables, for every* $k \leq N$.

Theorem IV-2.6: *For any* D *from* **BTEN**, *and every sufficiently large* N, *there is a command* E, *containing no symbols* $\mathcal{B} \cdots \mathcal{E}$, *such that* $D \sim_N E$.

The wording in **IV-2.6** is slightly ambiguous, so below is exactly what we mean. The corresponding ambiguous wordings in **2.7, 2.8** and **2.9** all mean that same order of quantifiers, that is, ℓ depends on D below, and, of course, E depends on N . (The language obtained by removing $\mathcal{B}\ldots\mathcal{E}$ from **BTEN** is often called the 'while'-language.)

IV-2.6: again : *For all D from* **BTEN**, *there exists a natural number ℓ, such that, for all $N \geq \ell$, there exists a command E, containing no symbols $\mathcal{B}\cdots\mathcal{E}$, such that $D \sim_N E$.*

Corollary IV-2.7: (to the proof of **2.6**) *For any D from* **ATEN**$+\mathcal{B}$, *and every sufficiently large N, there is a command E from* **ATEN** *such that $D \sim_N E$.*

Here **ATEN**$+\mathcal{B}$ is just **BTEN** with the **ite** command construction removed. This corollary has the same proof as **2.6**, simply omitting the case concerning **ite** commands in that proof.

Proof of IV-2.6: Assume the theorem to be false, and let D_0 be a command of shortest length such that one can find arbitrarily large N for which no command E failing to have any $\mathcal{B}\cdots\mathcal{E}$ in it can satisfy $E \sim_N D_0$. In the remainder of the proof, we eliminate the five possibilities for D_0 .

It cannot be atomic by Theorem 2, since we could just take E to be D_0 itself.

Suppose $D_0 = (D_1; D_2)$. Then we know, from the shortness of D_0, that for some ℓ and all $N \geq \ell$, there are commands E_1 and E_2 such that both $D_1 \sim_N E_1$ and $D_2 \sim_N E_2$, and so that $(E_1; E_2)$ has no $\mathcal{B}\cdots\mathcal{E}$ in it. It is easy to check that $(E_1; E_2) \sim_N D_0$, contradicting the basic assumption about D_0.

Both of the cases $D_0 = \mathbf{ite}(F)(D_1)(D_2)$ and $D_0 = \mathbf{whdo}(F)(D_1)$ are eliminated in the same straightforward manner.

Finally, suppose that $D_0 = \mathcal{B}_{i_1,\cdots,i_k} D_1 \mathcal{E}$. Then for some ℓ and and all $N \geq \ell$, there is an E_1 with no $\mathcal{B}\cdots\mathcal{E}$ in it such that we have $D_1 \sim_N E_1$. Let $m = \max\{\ell, i_1, \cdots, i_k\}$. Suppose $N \geq m$, and choose E_1 as above for that N. Finally let M be the maximum of N and the subscripts on all variables in E_1 . Now consider the command E_0 below:

$$(x_{M+1} \doteqdot x_{i_1} \ ; \ \cdots \ ; \ x_{M+k} \doteqdot x_{i_k} \ ; \ E_1 \ ; \ x_{i_1} \doteqdot x_{M+1} \ ; \ \cdots \ ; \ x_{i_k} \doteqdot x_{M+k}) \ .$$

The proof is completed by mechanically checking that $D_0 \sim_N E_0$. The main calculation is to check that, with respect to bins zero to "N", the command E_0 behaves as though it were the same as $\mathcal{B}_{i_1,\cdots,i_k} E_1 \mathcal{E}$. This is the content of the right-most assertion in each of the following three displays, which should be checked carefully. The other assertions are more-or-less obvious :
Fix $\underline{v}$ and $\underline{w}$ such that $v_i = w_i$ for all $i \leq N$. Then

$$||D_0||(\underline{v}) = err \;\Leftrightarrow\; ||D_1||(\underline{v}) = err \;\Leftrightarrow\; ||E_1||(\underline{w}) = err \;\Leftrightarrow\; ||E_0||(\underline{w}) = err \;.$$

Now suppose $||D_0||(\underline{v}) \neq err$. Fix $j \leq N$. If $j = i_p$ for some p, then

$$[||D_0||(\underline{v})]_j = v_{i_p} = w_{i_p} = [||E_0||(\underline{w})]_j \;.$$

If $j \neq i_p$ for any p, then

$$[||D_0||(\underline{v})]_j = [||D_1||(\underline{v})]_j = [||E_1||(\underline{w})]_j = [||E_0||(\underline{w})]_j \;.$$

Theorem IV-2.8: *For any C from* **BTEN**, *and every sufficiently large N, there is a command D from* **ATEN**$+\mathcal{B}$ *such that $C \sim_N D$.*

Proof. Assume the theorem to be false, and let C_0 be a command of shortest length such that one can find arbitrarily large N for which no command D failing to have any **ite** in it can satisfy $D \sim_N C_0$. The remainder of the proof consists in eliminating the five possibilities for C_0.

It cannot be atomic by **IV-2.2**, since we could just take D to be C_0 itself.

Suppose $C_0 = (C_1; C_2)$. Then we know, from the shortness of C_0, that for some ℓ and all $N \geq \ell$, there are commands D_1 and D_2 such that both $C_1 \sim_N D_1$ and $C_2 \sim_N D_2$, and so that $(D_1; D_2)$ has no **ite** in it. It is easy to check that $(D_1; D_2) \sim_N C_0$, contradicting the basic assumption about C_0.

Both of the cases $C_0 = \mathcal{B}_{i_1,\cdots,i_k} C_1 \mathcal{E}$ and $C_0 = \mathbf{whdo}(F)(C_1)$ are eliminated in the same straightforward manner.

It now follows that $C_0 = \mathbf{ite}(F_0)(C)(D)$. Furthermore, by the shortness of C_0, we know that C and D correspond to, for every sufficiently large N, **ite**-free commands C' and D' such that $C \sim_N C'$ and $D \sim_N D'$. Among all commands of the above form, with C and D as just specified, and such that one can find arbitrarily large N for which no command D_0 with $C_0 \sim_N D_0$ fails to have any **ite** in it, assume that the above command is one in which the quantifier-free formula F_0 is shortest possible. (At this point, C_0 itself

may not be shortest possible, but not to worry!) The proof is finished by eliminating all the four possibilities for F_0. By using our economical language with only two connectives from [**LM**], only the two atomic plus the two connective cases need be considered.

One case when F_0 is atomic is, for terms s and t, the command C_0 :

if $s < t$
thendo C
elsedo D

Fix some ℓ such that all variables occurring here, and all symbols $\mathcal{B}$, have subscripts no greater than ℓ, and such that, for all $N \geq \ell$, there are **ite**-free commands C'_N and D'_N such that $C'_N \sim_N C$ and $D'_N \sim_N D$.

Now we have $C_0 \sim_N D_0$ for the following command D_0 :

$(x_{N+1} \dot{\leftarrow} s$;
$x_{N+2} \dot{\leftarrow} t$;
$\mathcal{B}_{N+1,N+2}$ **while** $x_{N+1} < x_{N+2}$
do $(C'_N$; $x_{N+2} \dot{\leftarrow} x_{N+1})\mathcal{E}$;
while $\neg\, x_{N+1} < x_{N+2}$
do $(D'_N$; $x_{N+2} \dot{\leftarrow} x_{N+1} + 1)$)

It is clear that D_0 is **ite**-free, and is readily checked that $D_0 \sim_N C_0$.

This takes care of one case where F_0 is atomic. The other case is to change $s < t$ to $s \approx t$. This is treated analogously to the above. Details will be left to the reader.

Thus F_0 must be built as either $\neg F_1$, or as $F_1 \wedge F_2$. Since F_0 has minimal length, we can eliminate these as well, and complete the proof, as follows.

The formula F_0 cannot be $\neg F_1$, because then we could consider

$$\mathbf{ite}(F_1)(D)(C) \ .$$

The formula F_0 cannot be $F_1 \wedge F_2$, because then, as explained below, we could consider

$$\mathbf{ite}(F_1)(\mathbf{ite}(F_2)(C)(D))(D) \ .$$

[Recall that the command
if $F_1 \wedge F_2$
thendo A
elsedo B
has the same effect as
if F_1
thendo if F_2
thendo A
elsedo B
elsedo B *]*

So in this latter case we first note that, for the following command E, there is an ℓ such that, for every $N \geq \ell$, there is an **ite**-free command E' such that $E \sim_N E'$. This is because C and D both have that property, and because F_2 is shorter than $F_1 \wedge F_2$. Here is E :
if F_2
thendo C
elsedo D .
But now the same holds for
if F_1
thendo E
elsedo D ,
because E and D both have that property, and because F_1 is shorter than $F_1 \wedge F_2$. But this last command has exactly the same effect on any input as our original **ite**$(F_1 \wedge F_2)(C)(D)$, so the proof is finished.

Theorem IV-2.9: *For any C from* **BTEN**, *and every sufficiently large N, there is a command E from* **ATEN** *such that $C \sim_N E$.*

Corollary IV-2.10: (to **2.9** and **2.5**) *For any C from* **BTEN** *and any $k > 0$, there is a command E from* **ATEN** *such that C and E compute the same function of "k" variables.*

This last corollary is exactly what we have been aiming for, namely, that [f is $\mathcal{BC} \Longrightarrow f$ is $\mathcal{AC}$].

Remark. At first glance, **2.9** would appear to be a quick corollary to

2.7 and **2.8**. However, this seems not to work. But all we need do is go back to their proofs and more-or-less prove **2.7** and **2.8** simultaneously, to get a proof of **2.9**. This now makes **2.7** and **2.8** redundant, but it seemed simpler to work our way up gradually.

Sketch Proof of IV-2.9: As mentioned, we proceed just as in the proofs of **2.7** and **2.8**, but this time eliminating *both* the $\mathcal{B}\cdots\mathcal{E}$ and the **ite** commands.

Assume the theorem to be false, and let C_0 be a command of shortest length from **BTEN** such that one can find arbitrarily large N for which no command E from **ATEN** can satisfy $C_0 \sim_N E$. The remainder of the proof consists in eliminating the five possibilities for C_0.

It cannot be atomic by **2.2**, since we could just take E to be C_0 itself.

Suppose $C_0 = (C_1; C_2)$. Then we know, from the shortness of C_0, that for some ℓ and all $N \geq \ell$, there are commands E_1 and E_2 from **ATEN** such that both $C_1 \sim_N E_1$ and $C_2 \sim_N E_2$. It is easy to check that $(E_1; E_2) \sim_N C_0$, contradicting the basic assumption about C_0.

The case $C_0 = \mathbf{whdo}(F)(C_1)$ is eliminated in the same straightforward inductive manner.

The case $C_0 = \mathcal{B}_{i_1,\cdots,i_k} C_1 \mathcal{E}$ is eliminated with exactly the same argument as in the proof of **2.7** (AKA the proof of **2.6**.) The only point is to note that no **ite** occurred in the construction used there.

The case $C_0 = \mathbf{ite}(F)(C)(D)$ is eliminated in much the same way as in the proof of **2.8**. However, there we did use a $\mathcal{B}\cdots\mathcal{E}$-construction in one place: namely, when dealing with

if $s < t$
thendo C
elsedo D

The construction given for the command D_0 there may however be easily modified as follows, using the same minor idea which occurs in dealing with the case just above. First, for the given N and chosen C'_N, fix some $k \geq 2$ such that no variable subscript in C'_N is larger than $N + k$. Now define D_0 as follows :

$(x_{N+1} \leftarrow\!\!\div\ s\ ;$
$x_{N+2} \leftarrow\!\!\div\ t\ ;$
$x_{N+k+1} \leftarrow\!\!\div\ x_{N+1}\ ;$
$x_{N+k+2} \leftarrow\!\!\div\ x_{N+2}\ ;$
while $x_{N+1} < x_{N+2}$
do $(C'_N\ ;\ x_{N+2} \leftarrow\!\!\div\ x_{N+1})\ \ ;$
$x_{N+1} \leftarrow\!\!\div\ x_{N+k+1}\ ;$
$x_{N+2} \leftarrow\!\!\div\ x_{N+k+2}\ ;$
while $\neg\ x_{N+1} < x_{N+2}$
do $(D'_N\ ;\ x_{N+2} \leftarrow\!\!\div\ x_{N+1} + 1)\ )$

This completes the sketch proof.

Appendix A: Nitty-Gritty of Recursive Functions and Encoding.

To begin below, we give the definition of *primitive recursive function*, and repeat that of *recursive function* from [**LM**] .

Elsewhere, the latter is often called a *partial* recursive function. Since the unavoidability of dealing with partial functions has been heavily emphasized here, let's be brave and 'modernize' the terminology a bit. When we wish to point out that some recursive function is total, we'll say so explicitly. But beware: in many sources, the words "recursive function" are taken to mean "total recursive function".

Both (induction/recursion) and (inductive/recursive) are pretty much synonymous pairs in most contexts. It may seem unusual at first that the definition of "recursive function" below does not include any specific inductive aspect. This is partially ameliorated when we eventually show that any primitive recursive function is also recursive, which is not obvious from the definitions, despite the word usage. Now in fact, besides primitive recursion, there are other inductive procedures for defining functions, procedures considerably more subtle than primitive recursion. This was first realized by Ackermann in the 1920's when he found an historically significant example along these lines. In **IV-10** later, we give some details, and a **BTEN** command for computing that function. This Ackermann function points up the historical situation in the decades just before computability came to be understood, as is well explained in [**MeMeMe**]. For a time, there was considerable doubt about whether a single definition could exist which would capture every possibility of 'definition by induction', much less every possibility for computability. See also **IV-11** below for an elegant language of McCarthy, which puts induction/recursion back into the center of the picture in yet another general definition of computability.

Let's get on with the nitty-gritty! Although somewhat excessive notation, for uniformity, most functions below will be named with two or more capital Roman letters. The names are meant to be helpful to the reader's memory (or at least the author's), given that there are so many of these functions!

The four types of starter functions are
(i) the three functions $\mathbf{N} \times \mathbf{N} \to \mathbf{N}$ as follows: the addition function ADD ; the multiplication function $MULT$; the function $\overline{\chi}_{<}$, which maps (x, y) to 0 or 1 depending on whether $x < y$ or not ; as well as

(ii) all the projection functions $PJ_{n,k} : \mathbf{N}^n \to \mathbf{N}$, mapping $(a_1, \cdots, a_n)$ to a_k, for $1 \leq k \leq n$.

Define the operation of minimization, using ingredient F, where the domain of F is a subset $\mathbf{N}^{j+1}$ for some $j > 0$, producing a function g, taking the domain of g to be the following subset of $\mathbf{N}^j$:

$$\{(a_1, \cdots, a_j) : \exists k \text{ with } F(i, a_1, \cdots, a_j) \text{ defined for } 0 \leq i \leq k \text{ and } F(k, a_1, \cdots, a_j) = 0\}$$

and defining, for all $(a_1, \cdots, a_j)$ in domain(g) above,

$$g(a_1, \cdots, a_j) \ := \ \min\{k : F(k, a_1, \cdots, a_j) = 0\}.$$

Define the operation of composition, using ingredients $F, H_1, \cdots, H_j$, where we have: domain$(F) \subset \mathbf{N}^j$; for some ℓ and all s, domain$(H_s) \subset \mathbf{N}^\ell$; producing a function g whose domain is taken to be

$$\{(a_1, \cdots, a_\ell) \in \bigcap_{s=1}^{j} \text{domain}(H_s) : [H_1(a_1, \cdots, a_\ell), \ \ldots \ , H_j(a_1, \cdots, a_\ell)] \in \text{domain}(F)\} \ ;$$

and defining, for all $(a_1, \cdots, a_\ell)$ in the domain of g above,

$$g(a_1, \cdots, a_\ell) \ := \ F[H_1(a_1, \cdots, a_\ell), \ \ldots \ , H_j(a_1, \cdots, a_\ell)] \ .$$

This function g will be denoted below as $F \circ (H_1, \ \ldots \ , H_j)$, or just $F \circ H_1$ if $j = 1$.

Define the operation of primitive recursion , using ingredients $F : \mathbf{N}^k \to \mathbf{N}$ and $H : \mathbf{N}^{k+2} \to \mathbf{N}$ (both total functions!), producing a total function $g : \mathbf{N}^{k+1} \to \mathbf{N}$ as follows :

$$g(\vec{a}, 0) \ = \ F(\vec{a}) \ ;$$

$$g(\vec{a}, b+1) \ = \ H(\vec{a}, b, g(\vec{a}, b)) \ .$$

Henceforth we'll try to use the vector notation as above, with $\vec{a}$ meaning $(a_1, \cdots, a_k)$ for some k.

It's necessary to also consider vectors of length zero; that is, the case $k = 0$. Then the above display is interpreted as follows:

$$g(0) = \ell \quad \text{(any natural number)};$$
$$g(b+1) = H(b, g(b)) \ .$$

A *primitive recursive function* is any function $g : \mathbf{N}^k \to \mathbf{N}$ which can be built from the starter functions in finitely many steps using nothing more than compositions and primitive recursions. Since the starters are total and a composition of total functions is total, a primitive recursive function is indeed total. A fussier definition would refer to the existence of a 'primitive recursive verification column'; that is, a finite sequence of functions $g_1, \cdots, g_m = g$ for which each g_i is a starter, or is a composition using g_j's for $j < i$ as ingredients, or is a primitive recursion using g_j's for $j < i$ as ingredients. A simpler set of starter functions is often used in defining the set of primitive recursive functions, as discussed a few paragraphs below.

A *(partial) recursive function* (or just plain *recursive function*) is any function $g : D \to N$, where $D \subset \mathbf{N}^k$ for some k, which can be built from the starter functions in finitely many steps using nothing more than compositions and minimizations. Here again we could be more explicit, and introduce 'recursive verification columns'. For this, see **[LM]**, page 493.

A *total* recursive function may be defined to be any recursive function whose domain happens to be all of $\mathbf{N}^k$ for some k. (It is an interesting theorem, but irrelevant to this paper, that any total recursive function has a "verification column" consisting entirely of total functions—each minimization step uses a so-called "regular" function as ingredient. This is actually a very quick corollary of Kleene's Normal Form Theorem **IV-1.2**, occurring previously, but which 'logically' comes later, as explained earlier!)

In the opposite extreme, note that once we've shown that all constant functions are recursive, the function with empty domain (obtainable by minimizing any constant non-zero function) is in fact also recursive. It is the only recursive function for which the 'number of variables' is undefined. Another peripheral issue is that one can 'austerify' the definition of "recursive" by restricting to minimization of *relations*, total functions taking only 0 and 1 as values. This can also be deduced from **IV-1.2** (though the part about "taking only 0 and 1 as values" is rather trivial). You will notice below that only this sort of minimization step is used in the build-up.

The terminology introduced above would be manifestly stupid if it weren't true that any primitive recursive function is in fact a recursive function. But

it will be several paragraphs below before we get to the proof. So, in the meantime, there are a few places below where it behooves us to note explicitly that some function is both recursive and primitive recursive.

First here is an interesting (but peripheral) fact for which it seems worth giving the details: *the usual definition of primitive recursive uses a rather simpler set of starter functions, namely the constant function, $CT_{1,0}$, of one variable taking only the value 0; the successor function, SC , which adds 1 to any natural number; and all the projections $PJ_{n,i}$.*

Let PRIM' be the set of functions defined by these alternative starters, using composition and primitive recursion, and let PRIM be the set originally defined. To show inclusions each way, it clearly suffices to check that the starter functions in each case are in the other set.

For checking $\mathsf{PRIM}' \subset \mathsf{PRIM}$, let $CT_{1,\ell}$ more generally be the constant function of one variable taking only the value ℓ. Then

$$CT_{1,\ell}(0) \ = \ell \quad ;$$

$$CT_{1,\ell}(b+1) \ = \ PJ_{2,2}(b, CT_{1,\ell}(b)) \ .$$

So $CT_{1,\ell} \in \mathsf{PRIM}$. Also $SC \in \mathsf{PRIM}$, since

$$SC \ = \ ADD \circ (PJ_{1,1} \ , \ CT_{1,1}) \ ,$$

this being our fancy way of saying that $SC(b) = b + 1$.

For checking $\mathsf{PRIM} \subset \mathsf{PRIM}'$, first observe that

$$ADD(a, 0) \ = PJ_{1,1}(a) \quad ;$$

$$ADD(a, b+1) \ = \ SC \circ PJ_{3,3}(a, b, ADD(a, b)) \ .$$

So $ADD \in \mathsf{PRIM}'$. Also $MULT \in \mathsf{PRIM}'$, since

$$MULT(a, 0) \ = CT_{1,0}(a) \quad ;$$

$$MULT(a, b+1) \ = \ ADD \circ (PJ_{3,1}, PJ_{3,3})(a, b, MULT(a, b)) \ .$$

To deal with $\overline{\chi}_{<}$, we need to build up a small repertoire of other functions. In each case we'll check recursivity as well, later.

Let SG be the 'sign' function, which maps 0 to itself and all other natural numbers to 1. Reversing 1 and 0 above, its 'reversal', $\overline{SG}$, seems more

fundamental, since the identity $SG = \overline{SG} \circ \overline{SG}$ then takes care of SG. As for $\overline{SG}$, notice that

$$\begin{aligned} \overline{SG}(0) &= 1 \quad ; \\ \overline{SG}(b+1) &= CT_{1,0} \circ PJ_{2,1}(b, \overline{SG}(b)) \ . \end{aligned}$$

This shows $\overline{SG} \in \mathsf{PRIM}'$.

Let PD, the 'predecessor' function, map 0 to itself, and subtract 1 from all other numbers. Thus

$$\begin{aligned} PD(0) &= 0 \quad ; \\ PD(b+1) &= PJ_{2,1}(b, PD(b)) \ . \end{aligned}$$

This shows $PD \in \mathsf{PRIM}'$.

Let PS be 'proper subtraction', which we'll often write

$$PS(a,b) = a \dot{-} b := \begin{cases} a-b & \text{if } a \geq b \ ; \\ 0 & \text{if } a \leq b \ . \end{cases}$$

Then

$$\begin{aligned} PS(a,0) &= PJ_{1,1}(a) \quad ; \\ PS(a,b+1) &= PD \circ PJ_{3,3}(a,b,PS(a,b)) \ . \end{aligned}$$

This shows $PS \in \mathsf{PRIM}'$.

To finish showing $\mathsf{PRIM} \subset \mathsf{PRIM}'$ by proving that $\overline{\chi}_< \in PRIM'$, just write $\overline{\chi}_< = \overline{SG} \circ PS \circ (PJ_{2,2} \ , \ PJ_{2,1})$; that is, $a < b \iff b \dot{-} a > 0$.

In the course of establishing that primitive recursive functions are necessarily recursive, we must show directly that the functions dealt with above are also recursive, as follows:

To show $CT_{1,0}$ is recursive, note that

$$CT_{1,0}(a) = \min\{b : PJ_{2,2}(a,b) = 0\} \ .$$

By minimizing $\overline{\chi}_< \circ (CT_{1,0} \circ PJ_{2,2} \ , \ ADD \circ (PJ_{2,1}, PJ_{2,2}))$, we see that $\overline{SG}$ is recursive, since

$$\overline{SG}(b) = \min\{c : 0 < b + c\} \ .$$

Of course recursivity of SG itself follows, since it equals $\overline{SG} \circ \overline{SG}$, and that of $\chi_<$ because it equals $\overline{SG} \circ \overline{\chi}_<$

The successor function is recursive since

$$SC(a) \;=\; \min\{b : \overline{\chi}_<(a,b) = 0\} \;,$$

which tells us that SC is the minimization of $\overline{\chi}_< \circ (PJ_{2,2}, PJ_{2,1})$.

All the constant functions are recursive, since

$$CT_{1,1} = SC \circ CT_{1,0} \;\;,\;\; CT_{1,2} = SC \circ CT_{1,1} \;\;,\;\; CT_{1,3} = SC \circ CT_{1,2} \;\;,\;\; \text{etc.} \cdots ;$$

and $CT_{n,\ell} \;=\; CT_{1,\ell} \circ PJ_{n,1}$ (the constant function of "n" variables taking the value ℓ).

As for PS, just minimize $\overline{\chi}_< \circ (PJ_{3,2} \,,\; ADD \circ (ADD \circ (PJ_{3,3}, PJ_{3,1}) \,,\; CT_{3,1})))$, since

$$PS(a,b) \;=\; \min\{c : a < b + c + 1\} \;.$$

For PD, note that $PD(a) = PS(a,1)$, so $PD \;=\; PS \circ (PJ_{1,1}, CT_{1,1})$.

Before continuing to nitty-gritify, here are a few **exercises**:

Using composition and minimization, do the alternative starters (namely $CT_{1,0}$, SC , and all the $PJ_{n,i}$) generate *all* the recursive functions? (By what was done just above, they certainly generate nothing *but* recursive functions.)

If we use $\chi_<$ in place of $\overline{\chi}_<$ in the usual starters, do we get all the recursive functions?

Assuming the first question hasn't been answered "yes", and harking forward to the theorem that the set of total recursive functions is preserved by the operation of primitive recursion: Using composition, minimization *and* primitive recursion, do the alternative starters (namely $CT_{1,0}$, SC , and all the $PJ_{n,i}$) generate all the recursive functions?

Operators preserving the sets of primitive recursive and of recursive functions.

Next we show that our function sets are closed under definition by cases :

If h, g_1 and g_2 are primitive recursive (resp. recursive), then so is f, where

$$f(\vec{a}) \;:=\; \begin{cases} g_1(\vec{a}) & \text{if } h(\vec{a}) = 0 \;\; ; \\ g_2(\vec{a}) & \text{if } h(\vec{a}) > 0 \;\; . \end{cases}$$

(Often h is a relation and the second condition is written $h(\vec{a}) = 1$.)
The domain of f is taken to be

$$\text{domain}(g_1) \cap h^{-1}(0) \ \cup \ \text{domain}(g_2) \cap h^{-1}(\mathbf{N} \setminus 0) \ .$$

The result is immediate from the fact that

$$f(\vec{a}) \ = \ g_1(\vec{a})\overline{SG}(h(\vec{a})) + g_2(\vec{a})SG(h(\vec{a})) \ ,$$

that is,

$$f \ = \ ADD \circ (MULT \circ (g_1, \overline{SG} \circ h) \ , \ MULT \circ (g_2, SG \circ h)) \ .$$

Relations.

In this appendix, we shall make a notational distinction between a relation R defined as a subset of $\mathbf{N}^k$ for some k, and its characteristic function $\chi_{_R}$, which takes the value 1 on the set which is R, and 0 elsewhere. But after the appendix, we'll just abuse notation and use R for both the set and the function. So $R(\vec{a})$ is a way of saying $\vec{a} \in R$, and later it will be interchangeable with saying $R(\vec{a}) = 1$, though here that would be $\chi_{_R}(\vec{a}) = 1$ The relation and the set are defined to be recursive if and only if the function is, and similarly for primitive recursive.

A ubiquitous example below is R defined by $R(\vec{a}) \iff f(\vec{a}) = g(\vec{a})$, for given functions f and g. Then

$$R = \overline{SC} \circ ADD \circ (PD \circ (f, g) \ , \ PD \circ (g, f)) \ ,$$

so R is primitive recursive (or recursive), as long as f and g both are.

Now observe that our function sets are closed under propositional connectives applied to relations :

Suppose that R and S are primitive recursive (resp. recursive) relations. Then, with the obvious definitions (basically given in the proof),

$$\neg R \ \ , \ \ R \wedge S \ \text{ and } \ R \vee S$$

are also primitive recursive (resp. recursive).

We have

$$\chi_{\neg R} = \overline{SG} \circ \chi_{_R} \ ; \ \chi_{R \wedge S} = MULT \circ (\chi_{_R}, \chi_{_S}) \ ; \ \chi_{R \vee S} = SG \circ ADD \circ (\chi_{_R}, \chi_{_S}) \ .$$

Preservation under the 'bounded' operators :

For a given total function $f : \mathbf{N}^{k+1} \to \mathbf{N}$, define three more total functions on $\mathbf{N}^{k+1}$ as follows :

BSM_f, the *bounded sum*, by

$$BSM_f(\vec{a}, b) \ := \ \sum_{x<b} f(\vec{a}, x) \ := \ \sum_{x=0}^{b-1} f(\vec{a}, x) \ .$$

BPD_f, the *bounded product*, by

$$BPD_f(\vec{a}, b) \ := \ \prod_{x<b} f(\vec{a}, x) \ := \ \prod_{x=0}^{b-1} f(\vec{a}, x) \ .$$

BMN_f, the *bounded minimization*, by

$$BMN_f(\vec{a}, b) \ := \ \begin{cases} b & \text{if } \forall x < b \ , \ f(\vec{a}, x) \neq 0 \\ \min\{x : x < b \text{ and } f(\vec{a}, x) = 0\} & \text{otherwise} \end{cases} ;$$

A relation R on $\mathbf{N}^{k+1}$ gives *bounded quantifier relations* BUQ_R and BEQ_R in the obvious way:

$$BUQ_R(\vec{a}, b) \iff \forall x < b \ , \ R(\vec{a}, x) \ ,$$

and

$$BEQ_R(\vec{a}, b) \iff \exists x < b \ , \ R(\vec{a}, x) \ ,$$

Theorem. *These operations preserve the sets of primitive recursive functions, and of total recursive functions.*

Proof for primitive recursives. Clearly

$$BSM_f(\vec{a}, 0) \ = 0 \quad ;$$

$$BSM_f(\vec{a}, b+1) \ = \ BSM_f(\vec{a}, b) + f(\vec{a}, b) \ .$$

Thus BSM_f is obtained by primitive recursion from $CT_{k,0} : \mathbf{N}^k \to \mathbf{N}$, and the following function : $\mathbf{N}^{k+2} \to \mathbf{N}$:

$$ADD \circ (PJ_{k+2,k+2} \ , \ f \circ (PJ_{1,k+2}, PJ_{2,k+2}, \cdots, PJ_{k+1,k+2})) \ ,$$

since the latter maps $(\vec{a}, b, BSM_f(\vec{a}, b))$ to $BSM_f(\vec{a}, b) + f(\vec{a}, b)$.

It is immediate that BSM_f is primitive recursive, as long as f is.

The argument for BPD_f is very similar, using the fact that

$$BPD_f(\vec{a}, 0) \ = 1 \quad ;$$

$$BPD_f(\vec{a}, b+1) \ = \ BPD_f(\vec{a}, b) \cdot f(\vec{a}, b) \ .$$

Next note that

$$[\forall x < b \ , \ R(\vec{a}, x)] \iff \prod_{x<b} \chi_R(\vec{a}, x) = 1 \ .$$

Thus $\chi_{BUQ_R} = BPD_{\chi_R}$. So if R is a primitive recursive relation, i.e. if χ_R is a primitive recursive function, then so is χ_{BUQ_R} , as required.

Now

$$\exists x < b \ , \ R(\vec{a}, x) \iff \neg \forall x < b \ , \ \neg R(\vec{a}, x) \ .$$

Thus $BEQ_R = \neg BUQ_{\neg R}$, from which it is immediate that the set of primitive recursive functions is preserved also by the BEQ-operator.

Finally, given f, define a relation Z_f by

$$\chi_{Z_f} \ := \ \chi_= \circ (f, CT_{k+1,0}) \ .$$

That is,

$$Z_f(\vec{a}, b) \iff f(\vec{a}, b) = 0 \ .$$

So f primitive recursive implies that Z_f also is.

Next define a relation R_f by

$$\chi_{R_f}(\vec{a}, d) \ := \begin{cases} 0 & \text{if } \exists b < d+1 \ , \ f(\vec{a}, b) \neq 0 \ ; \\ 1 & \text{otherwise} \quad . \end{cases}$$

Clearly

$$\neg R_f(\vec{a}, d) \iff \exists x < d+1 \ , \ \neg Z_f(\vec{a}, x) \ ,$$

so

$$\chi_{R_f} = \overline{SG} \circ \chi_{BEQ_{\neg Z_f} \circ (\vec{a}, d) \mapsto (\vec{a}, d+1)} \ .$$

So from what is just above plus the BEQ-preservation result, we see that f primitive recursive implies that R_f also is.

But now

$$BMN_f(\vec{a}, b) \;=\; \sum_{x<b} \chi_{R_f}(\vec{a}, x) \;,$$

so $BMN_f = BSM_{\chi_{R_f}}$, which gives what is needed to complete the proof : f primitive recursive implies that BMN_f also is.

Anticipating proof that BSM and BPD preserve total recursives.
This result will not be used in the next several pages, until after we prove that primitive recursion preserves the set of total recursive functions. At that point, the result follows immediately from the primitive recursions for BSM and BPD given at the beginning of the previous proof.

Proof that BUQ, BEQ and BMN preserve the total recursives.
Because of the paragraph just above, this must proceed differently than the proof two paragraphs (and further) above.

For a total function f, we have

$$BMN_f(\vec{a}, b) = \min\{x : (x < b \wedge f(\vec{a}, x) = 0) \vee x = b\} = \min\{x : S(x, \vec{a}, b) = 0\} \;,$$

where $S = S_1 \vee S_2$, with

$$\chi_{S_2} = \chi_{=} \circ (PJ_{1,k+2}, PJ_{k+2,k+2})$$

and $S_1 = S_3 \wedge S_4$ with

$$\chi_{S_3} = \chi_{<} \circ (PJ_{1,k+2}, PJ_{k+2,k+2}) \quad \text{and} \quad S_4 = Z_f \text{ (from the previous proof)} \;.$$

From all this, it is clear that if f is total recursive, then so is BMN_f .

For a relation R on $\mathbf{N}^{k+1}$, define another one, S, on $\mathbf{N}^{k+2}$, by

$$S(x, \vec{a}, b) \iff R(\vec{a}, x) \;\vee\; x = b \;.$$

Then recursiveness of R implies that for S, which in turn implies recursiveness for the relation on $(\vec{a}, b) \in \mathbf{N}^{k+1}$ which is stated as "$\min\{x : S(x, \vec{a}, b)\} \;<\; b$". But the latter "statement" is just "$\exists x < b \;,\; R(\vec{a}, x)$", which is $BEQ_R(\vec{a}, b)$. Thus, recursiveness of R implies that for BEQ_R .

Finally note that $BUQ_R = \neg BEQ_{\neg R}$, dealing with BEQ, and completing the proof.

Often these "bounded operators" are applied in a slightly more general context, with the bound $x < b$ replaced by $x < h(\vec{a}, b)$. As long as h is also primitive recursive or recursive (as well as f), the resulting function is primitive recursive or recursive, respectively. This is easily seen by combining the previous theorem with closure under composition. For example, the function

$$g \ : \ (\vec{a}, b) \ \mapsto \ \min\{x : x < h(\vec{a}, b) \ , \ f(\vec{a}, x) = 0\}$$

(where $h(\vec{a}, b)$ replaces the right-hand side above when the latter is undefined), is just the function

$$g = BMN_f \circ (PJ_{1,k+1}, \cdots, PJ_{k,k+1} \ , \ h \circ (PJ_{1,k+1}, \cdots, PJ_{k+1,k+1})) \ ,$$

as is easily calculated.

Similarly for BSM, BPD, BUQ and BEQ.

We shall refer to applications of these more general principles also as *bounded minimization, bounded sum*, etc., in applications below.

Basic Coding and Gödel's Clever Function.

Temporarily use precursive to mean "both recursive and primitive recursive".

Define $QOT, REM : \mathbf{N} \times \mathbf{N}_+ \to \mathbf{N}$ by taking $QOT(a, d)$ and $REM(a, d)$ to be the quotient and remainder respectively when d is divided into a .

Thus,

$$QOT(a, d) = \min\{q : q < a + 1 \text{ and } a < d(1 + q)\}$$

and

$$REM(a, d) = a - dQOT(a, d) \ .$$

Applying bounded minimization and more basic stuff, QOT and REM are precursive if we arbitrarily set $QOT(a, 0) = a + 1$ and $REM(a, 0) = a$, to extend these functions to all of $\mathbf{N}^2$.

Another precursive function occurring later is ODD, where

$$ODD(b) \iff b \text{ is odd} \iff REM(b, 2) = 1 \ .$$

Precursivity follows quickly from the latter characterization.

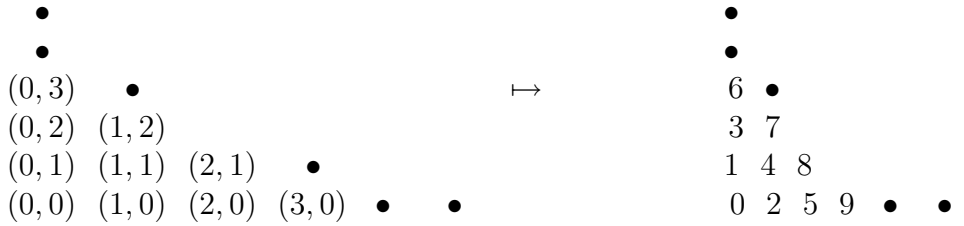
Define $CAN : \mathbf{N}^2 \to \mathbf{N}$ by $CAN(a,b) \ := \ a \ + \ \ (a+b)(a+b+1)/2$.

Then CAN is bijective and precursive.

Precursiveness is immediate from the formula, since

$$CAN(a,b) \ := \ a \ + \ \ QOT((a+b)(a+b+1),2) \ .$$

And bijectivity is clear from the famous Cantor diagram below (which motivates the formula).



Define $NAC_1, NAC_2 : \mathbf{N} \to \mathbf{N}$ to be the components of CAN^{-1} .

By definition, $CAN(NAC_1(c), NAC_2(c)) = c$ and $NAC_i(CAN(a_1,a_2)) = a_i$.

Thus, $NAC_1(c) = \min\{a < c+1 : \exists b < c+1 \text{ with } CAN(a,b) = c\}$,

and $NAC_2(c) = \min\{b < c+1 : \exists a < c+1 \text{ with } CAN(a,b) = c\}$.

So the NAC_i are precursive by the results in the last section.

By iterating CAN, we can get, for each $t \geq 2$, a precursive bijection

$$CAN_t \ : \ \mathbf{N}^t \ \to \ \mathbf{N} \ ,$$

whose inverse has precursive components. For example, CAN_4 could be chosen to be the composite

$$\mathbf{N}^4 = \mathbf{N}^2 \times \mathbf{N} \times \mathbf{N} \ \stackrel{CAN\times 1\times 1}{\longrightarrow} \mathbf{N} \times \mathbf{N} \times \mathbf{N} \ \stackrel{CAN\times 1}{\longrightarrow} \mathbf{N} \times \mathbf{N} \ \stackrel{CAN}{\longrightarrow} \mathbf{N} \ .$$

That is,

$$CAN_4(a,b,c,d) \ := \ CAN(CAN(CAN(a,b),c),d) \ .$$

Define $CLV : \mathbf{N}^2 \to \mathbf{N}$ by

$$CLV(c,i) \ := \ REM(NAC_1(c) \ , \ 1 + (i+1)NAC_2(c)) \ .$$

This is Gödel's clever function. Clearly from its definition and what we've just proved, the function CLV is precursive.

Theorem. (Gödel) $\forall n\ \forall (a_0, a_1, \cdots, a_n) \in \mathbf{N}^{n+1}$, $\exists c \in \mathbf{N}$ *with* $CLV(c, i) = a_i$ *for* $0 \leq i \leq n$, *and with* $c < h_n(a_0, a_1, \cdots, a_n)$ *for a suitable precursive function* h_n.

Also $CLV(c, i) < c$ *if* $c > 0$.

Proof. The second claim is clear, since $NAC_1(c) < c$ if $c \neq 0$ (look at the columns in the Cantor diagram).

For the first, let $c = CAN(a, b)$, where $b = n!(1 + \max_i\{a_i\})$, and where

$$a < (1 + b)(1 + 2b) \cdots (1 + (n + 1)b) \qquad \text{(which gives the function } h_n\text{)}$$

is chosen so that $REM(a\ ,\ 1 + (i + 1)b) = a_i$ for $0 \leq i \leq n$. By the Chinese Remainder Theorem, such an a exists, because

$$1 + b\ ,\ 1 + 2b\ ,\ \cdots\ ,\ 1 + (n + 1)b$$

are pairwise coprime. And thus,

$$CLV(c, i) = REM(NAC_1(c)\ ,\ 1+(i+1)NAC_2(c)) = REM(a\ ,\ 1+(i+1)b) = a_i\ ,$$

as required.

Define, for $n \geq 1$ and $(a_1, \cdots, a_n) \in \mathbf{N}^n$, the code of the latter sequence to be the number

$$< a_1, \cdots, a_n > \ :=\ \min\{\ c < h_n(n, a_1, \cdots, a_n)\ :$$
$$CLV(c, 0) = n \text{ and } CLV(c, i) = a_i \text{ for } 1 \leq i \leq n\ \}\ .$$

The existence of the code of course depends on the previous theorem.

Corollary. If $< a_1, \cdots, a_n > \ = \ < b_1, \cdots, b_m >$, then $n = m$ and $a_i = b_i$ for all i.

Proof. If c equals the displayed (equal) codes, then both sides of the required equalities are just $CLV(c, i)$ for $0 \leq i \leq n$.

Notation. We shall also denote by CTR or CTR_n the function from $\mathbf{N}^n$ to $\mathbf{N}$ sending $(a_1, \cdots, a_n)$ to $< a_1, \cdots, a_n >$. By the corollary,

$$CTR \ : \ \bigcup_{n \geq 1} \mathbf{N}^n \ \longrightarrow \ \mathbf{N}$$

is injective. Restricted to each $\mathbf{N}^n$, the function CTR is precursive, by bounded minimization and the formula defining $< a_1, \cdots, a_n >$.

Define CDE, a relation on $\mathbf{N}$, by

$$CDE(c) = 1 \iff c =< a_1, \cdots, a_n > \text{ for some } n \geq 1 \text{ and } a_i \in \mathbf{N} \ .$$

Exercise. Show that $CDE(c) = 0$ for $0 \leq c \leq 12$, but $CDE(13) = 1$, and $13 =< 0 >$.

Then

$$CDE(c) \iff \forall d < c \text{ we have [either } CLV(d,0) \neq CLV(c,0) \text{ or}$$

$$CLV(d,0) = CLV(c,0)(= n, \text{ say}) \text{ , and } \exists i \leq n \text{ with } CLV(d,i) \neq CLV(c,i)] \ .$$

Thus, combining the fact that CLV is precursive with several facts about bounded operators and connectives from the previous section, we see that CDE is precursive.

Proof that primitive recursives ARE recursive.
Now we'll be able to stop using that ugly last 'word' of the previous paragraph.

Theorem. *If, in the definition of the operation of primitive recursion, the ingredient (total) functions F and H are recursive, then the resulting function g is also total recursive.*

Corollary. *Every primitive recursive function is recursive.*
This follows inductively on the number of steps needed to build the function. Until this theorem, the only problematic case would have been when the last step is a primitive recursion; now nothing is problematic.

Proof of theorem. Fix $(\vec{a}, b)$ for now. For any c such that

$$(*) \text{ or } (*)_{\vec{a},b} \qquad CLV(c,k) = g(\vec{a},k) \quad \text{for } \ 0 \leq k \leq b \ \ ,$$

the equations in the definition of primitive recursion give

$$(**) \qquad CLV(c,0) = F(\vec{a}) \quad ;$$

$$(***) \qquad CLV(c,k+1) = H(\vec{a},k,CLV(c,k)) \quad \text{for } 0 \le k < b \ .$$

Conversely, for any c for which $(**) + (***)$ holds, an easy induction on k gives that $(*)$ holds.

Now define $J : \mathbf{N}^{k+1} \to \mathbf{N}$ by

$$J(\vec{a},b) \ := \ \min\{c : (*) \text{ holds}\} \ = \ \min\{c : (**) + (***) \text{ holds}\} \ .$$

Then J is total, since such a c in $(*)$ exists for every $(\vec{a},b)$, and is recursive since $(**) + (***)$ gives a set of recursive relations to minimize.

Define a now obviously total recursive function G by

$$G(\vec{a},b) \ := \ CLV(J(\vec{a},b),b) \ .$$

By $(*)_{\vec{a},b}$ [with $c = J(\vec{a},b)$ and $k = b$], we get

$$g(\vec{a},b) = CLV(c,b) = CLV(J(\vec{a},b),b) = G(\vec{a},b) \ .$$

This holds for all $(\vec{a},b)$, so $g = G$, and g is recursive, as required.

The soon-to-be-pervasive 'induction on history', AKA 'course-of-values recursion', tool.

Often a total function (such as the Fibbonacci sequence) is defined using several or all previous values, not just the immediately previous value. As long as it's just one coordinate we are dealing with, constructions like that preserve the set of primitive recursive functions. In more complicated cases of induction simultaneously on more than one coordinate ('nested recursions'), things can be more complicated (see Ackermann's function in Subsection IV-10). Such constructions of course preserve computability in the informal sense, and so must preserve the set of total recursive functions at least. Here we'll stick to one coordinate, which we take to be the last one, and prove preservation of both sets.

Beginning in a backwards direction, if g is any total function of "$k+1$" variables, define a new total function HLV_g, its 'history on the last variable',

as follows, using the coding to compile the complete previous history into a single number:

$$HLV_g(\vec{a},b) :=< g(\vec{a},0), g(\vec{a},1), \cdots, g(\vec{a},b-1) >= CTR(g(\vec{a},0), \cdots, g(\vec{a},b-1)) \ .$$

Since we are using various CTR's involving different numbers of variables, it is not completely obvious from this definition that the HLV-operator will preserve our function sets. Furthermore, the value when $b = 0$ must be interpreted, and we just take this to be

$$(\#) \qquad HLV_g(\vec{a},0) \ := \ 0 \ .$$

To deal with the problem, define a 'concatenation' function $CNCT$ by

$$CNCT(a,b) \ := \ \min\{ \ c \ : \ CLV(c,0) = CLV(a,0) + CLV(b,0) \ \wedge$$

$$\forall k < CLV(a,0) \ , \ CLV(c,1+k) = CLV(a,1+k) \ \wedge$$

$$\forall k < CLV(b,0) \ , \ CLV(c,1+k+CLV(a,0)) = CLV(b,1+k) \ \} \ .$$

The lengthy condition above is recursive and so $CNCT$ is recursive. It is also total from the basic Gödel theorem on his clever function CLV. The bound derived in that theorem to show CLV to be primitive can easily be parlayed into a bound for c above, and so the display can be interpreted as a bounded minimization, showing that $CNCT$ is actually primitive recursive. Finally, those conditions were set up, in view of the basic properties of CLV, to make it clear that $CNCT$ has the following property:

$$CNCT(< a_1, \cdots, a_n >, < b_1, \cdots, b_m >) \ = \ < a_1, \cdots, a_n, b_1, \cdots, b_m > \ .$$

Now we can write

$$(\#\#) \qquad HLV_g(\vec{a}, b+1) \ = \ CNCT(HLV_g(\vec{a},b) \ , \ < g(\vec{a},b) >) \ .$$

Combined with (#), this expresses HLV_g as a primitive recursion, so the HLV-operator does indeed preserve the sets of primitive recursive and of total recursive functions.

As mentioned, this is really backwards. What we mostly want is to prove that a g expressed recursively in terms of its history is itself recursive (not merely to assume it recursive and prove its history also to be).

So here is a 'course-of-values recursion' for g :

$$g(\vec{a}, b) \;=\; F(\vec{a}, b, HLV_g(\vec{a}, b)) \ .$$

One assumes F is a given function of "$k+2$" variables. The above display then defines the function g uniquely, since, with increasing b, we have $g(\vec{a}, 0) \;=\; F(\vec{a}, 0, 0)$, then $g(\vec{a}, 1) \;=\; F(\vec{a}, 1, < g(\vec{a}, 0) >)$, then $g(\vec{a}, 2) \;=\; F(\vec{a}, 2, < g(\vec{a}, 0), g(\vec{a}, 1) >)$, etc.

Theorem. *If F* (in the definition of course-of-values recursion) *is primitive recursive (respectively total recursive), then so is g.*

Proof. Just rewrite (##) in the form

$$HLV_g(\vec{a}, b+1) \;=\; CNCT(HLV_g(\vec{a}, b) \ , \ \ < F(\vec{a}, b, HLV_g(\vec{a}, b)) >) \ .$$

Combined with (#), this now gives a primitive recursion characterization of HLV_g using only F, not g. It also uses that $c \mapsto < c >$ is primitive recursive. Thus HLV_g is primitive recursive (respectively total recursive). Now from the display defining "course-of-values recursion for g", the latter function g is also primitive recursive (respectively total recursive), as required.

At this point, it is time to retract the phrase "nitty-gritty", to the extent that it might be taken pejoratively. The buildup of theory in this appendix is certainly entirely elementary, in the sense that it stands alone, independent of any other non-trivial mathematics (except perhaps Cantor's D minor work!) But it is itself by this time pretty non-trivial, and has significant applications, as was explained in Subsection IV-1, and as will also occur in Section VI.

Much of the work in the latter, and all of it in the next subsection, consists of showing a whole lot of relations to be primitive recursive. Course-of-values recursion is the principal tool. This is done relatively informally, without getting into details of converting a 'recursive description' of the relation into a recursive function F as above, to be used in the course-of-values recursion. Just below we do this for a simple example, the famous Fibbonaci sequence, and then also do it for an example more analogous to the later relevant examples, hopefully leaving the reader confident that she can supply the similar details, if challenged, in all the other applications.

The Fibbonaci sequence is the function $g : \mathbf{N} \to \mathbf{N}$ defined by

$$g(0) = 1 = g(1) \qquad \text{and} \qquad g(b) = g(b-1) + g(b-2) \quad \text{for } b \geq 2 \ .$$

To see this explicitly as a course-of-values recursion, we need a function F such that

$$g(b) \;=\; F(b, HLV_g(b)) \ .$$

But

$g(b-1) + g(b-2) =$

$$CLV(< g(0), \cdots, g(b-1) >, b) + CLV(< g(0), \cdots, g(b-1) >, b-1)$$
$$= CLV(HLV_g(b), b) + CLV(HLV_g(b), b-1) \ .$$

So, when $b \neq 0$ or 1, we may take F to behave as the function

$$ADD \circ (CLV \circ (PJ_{2,2} \ , \ PJ_{2,1}) \ , \ CLV \circ (PJ_{2,2} \ , \ PS \circ (PJ_{2,1}, CT_{2,1}))) \ ,$$

that is

$$F(b, a) = CLV(a, b) + CLV(a, b-1) \quad \text{for} \ \ b \geq 2 \ ;$$

and take

$$F(b, a) = 1 \quad \text{for} \ \ b < 2.$$

Using definition by cases, it is clear that F is primitive recursive. Thus g is also.

The main thing to notice is that you use CLV to get back, from the history as a single number, to any collection of particular earlier values of the function appearing in the recursive description.

In the next, final section of this appendix, we 'show' that certain 1-ary relations (or *predicates*) TRM, FRM and FRM_{free} are primitive recursive. All the work in the section following this appendix, and much of the work in Section VI, relates to similar, often more complicated, examples. Here is a simpler, but very similar example, done in all detail as a course-of-values recursion. See also the remarks after the analysis of TRM in the following section.

Suppose we have a 1-ary relation R for which $\neg R(0)$, and we define another one, S as follows:

$$S(a) \iff R(a) \text{ and [either } S(CLV(a, 1) \text{ or } S(CLV(a, 3) \text{ or both] } .$$

This clearly defines S since, for $a > 0$, both $CLV(a,1)$ and $CLV(a,3)$ are less than a. Identifying R and S with their characteristic functions (drop the χ's!), this can be written

$$S(a) = 1 \iff R(a) = 1 \text{ and [either } S(CLV(a,1)) = 1 \text{ or } S(CLV(a,3)) = 1] \, .$$

Equivalently,

$$S(a) = R(a) \cdot SG(S(CLV(a,1)) + S(CLV(a,3))) \, .$$

To write S as a course-of-values recursion in terms of R, we must find a function F such that

$$S(a) = F(a, HLV_S(a)) = F(\ a\ ,\ \ < S(0), S(1), \cdots, S(a-1) > \ ) \, .$$

Comparing the right-hand sides, the following will do:

$$F(a,b) = R(a) \cdot SG(CLV(b, CLV(a,1)) + CLV(b, CLV(a,3))) \, .$$

More formally

$$F = MULT \circ (R \circ PJ_{2,1}\ ,\ SG \circ ADD \circ$$

$$(CLV \circ (PJ_{2,2}\ ,\ CLV \circ (PJ_{2,1}, CT_{2,1}))\ ,\ CLV \circ (PJ_{2,2}\ ,\ CLV \circ (PJ_{2,1}, CT_{2,3})))) \, .$$

So this does the job, and therefore shows that S would be primitive recursive as long as R was. And you can see why no one ever wants to write down all the details for examples like this, as we do not do about 30 times over the next 60 pages !

Codes for terms and formulas.

Define inductively

$$\#_{\mathrm{trm}} \ : \ \{\text{set of terms in } 1^{\underline{\text{st}}}\text{order number theory}\} \ \longrightarrow \ \mathbf{N}$$

by

$$\begin{aligned} \#(0) \ &:= \ < 0_{\mathbf{N}}, 0_{\mathbf{N}}, 0_{\mathbf{N}} > \ ; \\ \#(1) \ &:= \ < 0_{\mathbf{N}}, 1_{\mathbf{N}}, 0_{\mathbf{N}} > \ ; \\ \#(x_j) \ &:= \ < 0_{\mathbf{N}}, j, j+1 > \ ; \\ \#(s+t) \ &:= \ < 1_{\mathbf{N}}, \#(s), \#(t) > \ ; \\ \#(s \times t) \ &:= \ < 2, \#(s), \#(t) > \ . \end{aligned}$$

Then, clearly, $\#(s) = \#(t)$ implies $s = t$.

Usually the subscripts $\mathbf{N}$ on 0 and 1 are suppressed. I just felt like re-emphasizing the distinction between numbers on the one hand, and constant symbols/terms in the formal language on the other.

We shall be using 'syntactic' versions, "a", of natural numbers a. These are terms in the formal language obtained by adding exactly that many '1' symbols, with suitable brackets, preceded by adding a '0' symbol. More precisely, here is the inductive definition :

$$\text{“}0_{\mathbf{N}}\text{”} \ := \ 0 \quad \text{and} \quad \text{“}a +_{\mathbf{N}} 1_{\mathbf{N}}\text{”} \ = \ \text{“}a\text{”} + 1 \ .$$

It's necessary to show that the function $SYN : \mathbf{N} \to \mathbf{N}$, which is given by $SYN(a) = \#_{TRM}(\text{“}a\text{”})$, is a primitive recursive function. Here's a definition of it by primitive recursion (temporarily again being careful about the distinction between numbers [semantics!] and symbols [syntax!]) :

$$SYN(0_{\mathbf{N}}) \ = \ \#_{TRM}(\text{“}0_{\mathbf{N}}\text{”}) \ = \ \#_{TRM}0 \ = \ < 0_{\mathbf{N}}, 0_{\mathbf{N}}, 0_{\mathbf{N}} > \ = \ CTR(0_{\mathbf{N}}, 0_{\mathbf{N}}, 0_{\mathbf{N}}) \, ,$$

and

$$SYN(a +_{\mathbf{N}} 1_{\mathbf{N}}) \ = \ \#_{TRM}(\text{“}a +_{\mathbf{N}} 1_{\mathbf{N}}\text{”}) \ = \ \#_{TRM}(\text{“}a\text{”} + 1) \ =$$

$$< 1_{\mathbf{N}}, \#_{TRM}(\text{“}a\text{”}), < 0_{\mathbf{N}}, 1_{\mathbf{N}}, 0_{\mathbf{N}} >> \ = \ CTR(1_{\mathbf{N}}, SYN(a), CTR(0_{\mathbf{N}}, 1_{\mathbf{N}}, 0_{\mathbf{N}})) \ .$$

Define TRM, a relation on $\mathbf{N}$, by

$$TRM(a) = 1 \iff \exists \text{ a term } t \text{ with } a = \#(t) \ .$$

Then

$$TRM(a) \iff CDE(a) \text{ and } CLV(a,0) = 3 \text{ and one of :}$$

$$CLV(a,1) = 0 \text{ and either } CLV(a,3) = 0 \text{ and } CLV(a,2) \leq 1$$
$$\text{or } CLV(a,3) = 1 + CLV(a,2) \ ;$$

or

$$CLV(a,1) = 1 \text{ and } TRM(CLV(a,2)) \text{ and } TRM(CLV(a,3)) \ ;$$

or

$$CLV(a,1) = 2 \text{ and } TRM(CLV(a,2)) \text{ and } TRM(CLV(a,3)) \ .$$

Thus TRM is primitive recursive.

Notice that this uses the fact that $CLV(a,i) < a$, since $a = 0$ doesn't occur, so that the above set of conditions constitute a generalized inductive characterization of TRM. This sort of argument for primitive recursivity will occur many times below without mention, particularly here and in Sections IV-3 and VI-1. The main part of the formal argument is a course-of-values recursion. A simplified example is presented in all detail at the end of the previous section of this appendix.

Define inductively

$$\#_{\text{frm}} \ : \ \{\text{set of formulas in } 1^{\underline{\text{st}}}\text{order number theory}\} \ \longrightarrow \ \mathbf{N}$$

by

$$\begin{aligned}
\#(s \approx t) &:= \ < 1, \#_{\text{trm}}(s), \#_{\text{trm}}(t) > \ ; \\
\#(s < t) &:= \ < 2, \#_{\text{trm}}(s), \#_{\text{trm}}(t) > \ ; \\
\#(\neg F) &:= \ < 3, \#_{\text{frm}}(F), \#_{\text{frm}}(F) > \ ; \\
\#(F \wedge G) &:= \ < 4, \#_{\text{frm}}(F), \#_{\text{frm}}(G) > \ ; \\
\#(\forall x_j F) &:= \ < 5, j, \#_{\text{frm}}(F) > \ .
\end{aligned}$$

Then, clearly, $\#(F) = \#(G)$ implies $F = G$.

Define FRM, a relation on $\mathbf{N}$, by

$$FRM(b) = 1 \iff \exists \text{ a formula } F \text{ with } b = \#(F) \ .$$

Then

$$FRM(b) \iff CDE(b) \text{ and } CLV(b,0) = 3 \text{ and one of :}$$

$$CLV(b,1) = 1 \text{ and } TRM(CLV(b,2)) \text{ and } TRM(CLV(b,3) \ ;$$

or

$$CLV(b,1) = 2 \text{ and } TRM(CLV(b,2)) \text{ and } TRM(CLV(b,3) \ ;$$

or

$$CLV(b,1) = 3 \text{ and } FRM(CLV(b,2)) \text{ and } CLV(b,2) = CLV(b,3) \ ;$$

or

$$CLV(b,1) = 4 \text{ and } FRM(CLV(b,2)) \text{ and } FRM(CLV(b,3) \ ;$$

or

$$CLV(b,1) = 5 \text{ and } FRM(CLV(b,3) \ .$$

Thus FRM is primitive recursive.

Define FRM_{free}, a relation on $\mathbf{N}$, by

$$FRM_{\text{free}}(b) = 1 \iff \exists F, \text{ a formula with no quantifiers , with } b = \#(F) \ .$$

Then we have a similar technical characterization of this relation, proving it to be primitive recursive. Just remove the last line (which starts with $CLV(b,1) = 5$); and change FRM to FRM_{free} in the other three places.

IV-3. Kleene's computation relation is primitive recursive.

This is a lengthy, but entirely elementary, exercise in building up various relations, and a few other functions, and proving them all primitive recursive, using the standard methods from Appendix A. Here is the complete list needed, followed by a dependency diagram, so you can see where we are headed. The first six were already dealt with in that appendix.

ODD – 'is an odd number'

CTR – Cantor

CDE – 'is a code number'

CLV – Gödel's clever function

TRM – 'is a term number'

FRM_{free} – 'is a (quantifier-free) formula number'

COM – 'is a command number'

MRK – 'is a marked command number'

ELN – evaluation of a term , substituting for variables

TRU – 'is true'

OUT – output next bin values

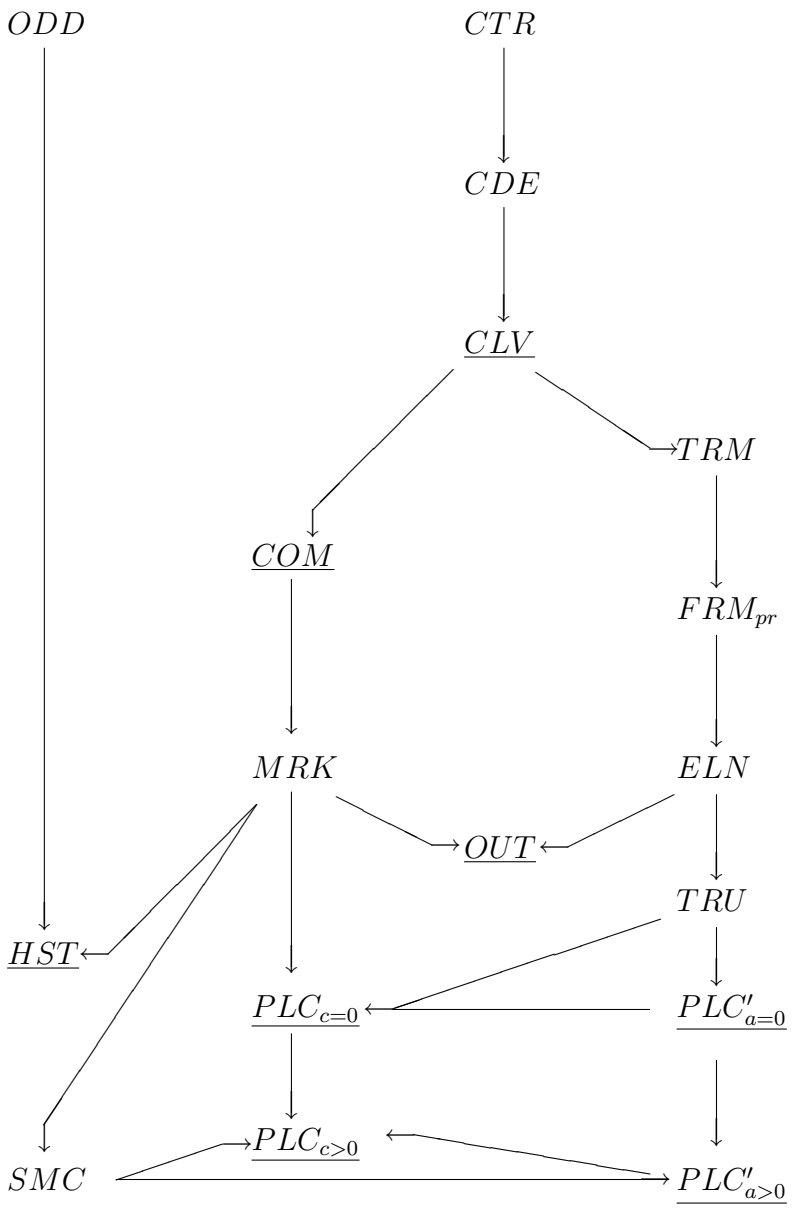
SMC – 'underlies the same command'

HST – 'is a history, perhaps'

PLC – the place to put the mark on an atomic command

KLN – Kleene (his computation relation)

$PRINT$ – singles out the computed function value



The definition of KLN directly involves the underlined relations.

For the rest of the section, we shall work our way down that diagram, writing any new coding conventions not already in Appendix A, writing the definition of each relation, and then re-expressing it in a form which makes it obvious that it is recursive, by use of the results in Appendix A. Recall that we'd just write, for example, $TRM(b)$ instead of $TRM(b) = 1$ (which means that b is the code of some term in the language of $1^{\underline{\text{st}}}$order number theory). The "$= 1$" is systematically suppressed. Again, informally, $REL(a, b, c) = 1$, or just $REL(a, b, c)$, mean that the relation called REL holds for the triple (a, b, c) of natural numbers.

Note that each coding given below does give different code numbers to different objects. For example, with codes for commands,

$$\#_{COM}(C) \;=\; \#_{COM}(D) \;\;\Longrightarrow\;\; C = D \;.$$

Usually the left-hand side above would just be written $\#C$. The more elaborate specification with the subscript is useful only in a few places where codes of several different types of objects are considered simultaneously.

Recall also from the previous subsection that we are working with **ATEN**, not **BTEN**, so there are only three types of commands.

COM: Commands are coded inductively as follows :

$$\#_{COM}(x_j \doteqdot s) \;:=\; < 1, j, \#_{TRM} s > \;;$$

$$\#_{COM}(D; E) \;:=\; < 2, \#_{COM} D, \#_{COM} E > \;;$$

$$\#_{COM}\mathbf{whdo}(F)(D) \;:=\; < 3, \#_{FRM} F, \#_{COM} D > \;.$$

Recall from Appendix A that $<\;,\;,\;>: \mathbf{N}^3 \to \mathbf{N}$ is the restriction of CTR, and that $\#_{TRM} s$ and $\#_{FRM} F$ are the code numbers of the term s and of the (quantifier-free) formula F.

Recall also that $CDE(c)$ means that c is the code number of some tuple of natural numbers, that is, for some $t > 0$ and $v_i \in \mathbf{N}$, we have

$$c =< v_1, v_2, \cdots, v_t >= CTR(v_1, v_2, \cdots, v_t) \;.$$

Then, Gödel's clever function CLV recovers the tuple from its code as follows: $CLV(c, 0) = t$ and $CLV(c, i) = v_i$ for $1 \leq i \leq t$.

All these functions just above have been proved to be primitive recursive in Appendix A.

Thus, since, by definition,

$$COM(c) \qquad \Longleftrightarrow \qquad c \text{ is the code of some command ,}$$

we have the following, recalling that
$FRM_{\text{free}}(c) \iff c$ is the code of a quantifier-free formula :

$$\underline{COM(c)} \qquad \Longleftrightarrow \qquad CDE(c) \text{ and } CLV(c,0) = 3 \text{ and one of}$$

$$CLV(c,1) = 1 \text{ and } TRM(CLV(c,3)) \text{ ; or}$$

$$CLV(c,1) = 2 \text{ and } COM(CLV(c,2)) \text{ and } COM(CLV(c,3)) \text{ ; or}$$

$$CLV(c,1) = 3 \text{ and } FRM_{\text{free}}(CLV(c,2)) \text{ and } COM(CLV(c,3)) \text{ .}$$

This shows COM to be primitive recursive, since we also proved TRM and FRM_{free} to be primitive recursive in Appendix A.

We shall now define "*marked command* " mathematically. This will be done inductively, just as we originally defined commands, that is, $\cdots$ defined the syntax of the language **ATEN**. To be very strict, we'd introduce some new symbol which will occur exactly once in the string which is a given marked command, either just before or just after (one or the other choice would be made for the definition) the occurrence of an atomic subcommand. The latter would be the subcommand which is marked. Removing the mark (shortening the string by one) then produces what we call the *underlying command*.

Since I cannot make up my mind between "before" and "after", and, more seriously, since it would be nice to remain picturesque about this, what we'll do here in print is to surround the marked subcommand with a rectangle. (You might even think of the rectangle as representing some sort of *read-write head* !) The fussy reader can erase 3/4 of each of these rectangles and take the 4th (vertical) side of it as the new symbol referred to above, either on the left or on the right, as you prefer.

Definition. A *marked command* is any finite string which can be generated by the following rules :

(i) $\boxed{x_i \dot{\leftarrow} t}$ is a marked command each term t in the assertion language and for each variable x_i .

$(\text{ii})_{\text{left}}$ $(C; D)$ is a marked command, for any marked command C and any command D .

$(\text{ii})_{\text{right}}$ $(C; D)$ is a marked command, for any command C and any marked command D .

(iii) $\mathbf{whdo}(F)(C)$ is a marked command, for each formula F in the assertion language, and all marked commands C.

MRK: Marked commands are coded inductively as follows :

$$\#_{MRK}(\ \boxed{x_j \dot{\leftarrow} s}\ ) \ := \ < 1, j, \#_{TRM} s > \ ;$$
$$\#_{MRK}(D; E) \ := \ < 2, \#_{MRK} D, \#_{COM} E > \quad \text{if mark is inside } D \ ;$$
$$\#_{MRK}(D; E) \ := \ < 4, \#_{COM} D, \#_{MRK} E > \quad \text{if mark is inside } E \ ;$$
$$\#_{MRK}\mathbf{whdo}(F)(D) \ := \ < 3, \#_{FRM} F, \#_{MRK} D > \ .$$

Note that the "4" on line 3 is the only change from defining the code of a 'just-plain command'.

Thus, since, by definition,

$$MRK(m) \qquad \Longleftrightarrow \qquad m \text{ is the code of some marked command ,}$$

we have the following :

$$\underline{MRK(m)} \qquad \Longleftrightarrow \qquad CDE(m) \text{ and } CLV(m,0) = 3 \text{ and one of :}$$
$$CLV(m,1) = 1 \ \text{ and } \ TRM(CLV(m,3)) \ ; \text{ or}$$
$$CLV(m,1) = 2 \ \text{ and } \ MRK(CLV(m,2)) \quad \text{and } \ COM(CLV(m,3)) \ ; \text{ or}$$
$$CLV(m,1) = 4 \ \text{ and } \ COM(CLV(m,2)) \quad \text{and } \ MRK(CLV(m,3)) \ ; \text{ or}$$
$$CLV(m,1) = 3 \ \text{ and } \ FRM_{\text{free}}(CLV(m,2)) \quad \text{and } \ MRK(CLV(m,3)) \ .$$

This shows MRK to be primitive recursive.

SMC:
By definition, the relation SMC is the following :

$$SMC(m,c) \iff m \text{ is the code of a marked command ,}$$
$$\text{and } c \text{ is the code of its underlying command .}$$

Thus we have :

$$\underline{SMC(m,c)} \iff MRK(m) \text{ and } COM(c) \text{ and one of :}$$

$$CLV(m,1) = 1 = CLV(c,1) \text{ and } m = c \text{ ; or}$$

$$CLV(m,1) = 2 = CLV(c,1) \text{ and } SMC(CLV(m,2), CLV(c,2))$$
$$\text{and } CLV(m,3) = CLV(c,3) \text{ ; or}$$

$$CLV(m,1) = 4 = 2CLV(c,1) \text{ and } SMC(CLV(m,3), CLV(c,3))$$
$$\text{and } CLV(m,2) = CLV(c,2) \text{ ; or}$$

$$CLV(m,1) = 3 = CLV(c,1) \text{ and } CLV(m,2) = CLV(c,2))$$
$$\text{and } SMC(CLV(m,3), CLV(c,3)) \text{ .}$$

This shows SMC to be primitive recursive.

ELN:
By definition,

$$ELN(a,b,c) \iff a \text{ is the code of a term } t,\ b =< v_0, \cdots, v_n > \text{ for}$$

$$\text{some } v_i \text{ and } n, \text{ and } c = t^{(v_0, v_1, \cdots, v_n,\ \underline{0})} .$$

That is, from [**LM**], page 211, the number c is obtained by substituting the numbers v_i for the variables x_i in the term whose code is a (and $0_{\mathbf{N}}$ for x_i if $i > n$, and, of course, substituting the actual numbers $0_{\mathbf{N}}$ and $1_{\mathbf{N}}$ for the constant symbols 0 and 1, and the actual operations on $\mathbf{N}$ for the function symbols $+$ and $\times$. The subscripts on $0_{\mathbf{N}}$ and $1_{\mathbf{N}}$ are suppressed almost everywhere in this section.)

Thus we have the following :

$$\underline{ELN(a,b,c)} \iff TRM(a) \text{ and } CDE(b) \text{ and one of :}$$

$$CLV(a,1) = 0\ ,\ CLV(a,3) = 0 \text{ and } CLV(a,2) = c\ ;$$

[This is the case where term#a is 0 or 1
(with codes resp. $< 0_{\mathbf{N}}, 0_{\mathbf{N}}, 0_{\mathbf{N}} >$ and $< 0_{\mathbf{N}}, 1_{\mathbf{N}}, 0_{\mathbf{N}} >$)
and so c is $0_{\mathbf{N}}$ or $1_{\mathbf{N}}$—we've temporarily reminded you
of the distinction between numbers and constant symbols here,
but usually won't use the subscripts.]

$$\text{or } CLV(a,1) = 0\ ,\ CLV(a,3) > 0\ ,$$

$$CLV(b,0) > CLV(a,2) \text{ and } c = CLV(b, CLV(a,2))\ ;$$

[This is the case where term#a is x_j
(with code $< 0, j, j+1 >$)
where $j = CLV(a,2) \leq n = CLV(b,0) - 1$, so $c = v_j$.]

$$\text{or } CLV(a,1) = 0\ ,\ CLV(a,3) > 0\ ,\ CLV(b,0) \leq CLV(a,2) \text{ and } c = 0_{\mathbf{N}}\ ;$$

[This is the case where term#a is x_j where $j > n$.]

$$\text{or } CLV(a,1) = 1 \text{ and } \exists\ c' \text{ and } c'' \text{ with } c = c' + c'' ,$$

$$\text{and } ELN(CLV(a,2), b, c') \text{ and } ELN(CLV(a,3), b, c'')\ ;$$

(This is the case where term$\#a$ is a sum of two terms—recall that $\#(s' + s'') =< 1, \#s', \#s'' >$. Also note that the existential quantifier is bounded—really we have the 'or'ing of "$c + 1$" conditions here, not really a quantifier.)

or $\quad CLV(a, 1) = 2$ and $\exists\ c'$ and c'' with $c = c'c''$,

and $\ ELN(CLV(a, 2), b, c')$ and $ELN(CLV(a, 3), b, c'')$.

(This is the case where term$\#a$ is a product of two terms—recall that $\#(s' \times s'') =< 2, \#s', \#s'' >$.)

This shows ELN to be primitive recursive.

OUT:

By definition,

$OUT(a, b, c) \iff a$ and c are codes of $(n+1)-$tuples for the same n, b is the code of a marked command M, and tuple$\#c$ is obtained by applying the marked atomic sub-command, say $\boxed{x_j \div s}$, within M to tuple$\#a$.

More precisely, as long as $j \leq n$, from the semantics of 1st order logic as in Ch. 6 of [**LM**] ,

if $a =< v_0, \cdots, v_n >$, then $c =< v_0, \cdots, v_{j-1}, s^{(v_0, \cdots, v_n, \underline{0})}, v_{j+1}, \cdots, v_n >$.

Thus we have the following :

$\underline{OUT(a, b, c)} \qquad \iff \qquad CDE(a)$, $CDE(c)$ and $MRK(b)$, and one of :

[$CLV(a, 0) = CLV(c, 0)$ actually follow from these.]

$CLV(b, 1) = 1$, $CLV(b, 2) < CLV(c, 0)$, $CLV(a, i) = CLV(c, i)$ for all

$i \neq CLV(b, 2)$ and $ELN(a, CLV(b, 3), CLV(c, CLV(b, 2) + 1))$;

[This is the case where the entire marked command$\#b$
is $\boxed{x_j \div s}$ with $j = CLV(b, 2) + 1 \leq n$ and $\#s = CLV(b, 3)$.
For $i \neq j$, the ith slots $CLV(a, i)$ and $CLV(c, i)$

of tuple#a and tuple#c are the same.
But the $(j+1)$th slot of tuple#c,
namely $CLV(c, j+1) = CLV(c, CLV(b,2)+1)$
is obtained by evaluating term#s using tuple#a .]

$$\text{or} \quad CLV(b,1) = 1 \ , \ CLV(b,2)(c,0) \ \text{ and } \ c = a \ \ ;$$

[This is the case where the entire marked command#b
is $\boxed{x_j \div s}$ but $j = CLV(b,2)+1 > n$.]

$$\text{or} \quad CLV(b,1) = 2 \ \text{ and } \ OUT(a, CLV(b,2), c) \ \ ;$$

[Here marked command#b is $(D;E)$ with the marked
command within D, and $\#_{MRK} D = CLV(b,2)$.]

$$\text{or} \quad CLV(b,1) = 4 \ \text{ and } \ OUT(a, CLV(b,3), c) \ \ ;$$

[Here marked command#b is $(D;E)$ with the marked
command within E, and $\#_{MRK} E = CLV(b,3)$.]

$$\text{or} \quad CLV(b,1) = 3 \ \text{ and } \ OUT(a, CLV(b,3), c) \ \ ;$$

[Here marked command#b is $\mathbf{whdo}(F)(C)$ and $\#_{MRK} C = CLV(b,3)$.]

This shows OUT to be primitive recursive.

TRU:

By definition,

$TRU(b,c) \iff$ for some $n \geq 0$, b is the code of an $(n+1)-$tuple $\vec{v} = (v_0, \cdots, v_n)$, and c is the code of a **quantifier-free** formula F such that F is true at $(\vec{v}, \underline{0})$.

Thus we have the following :

$$\underline{TRU(b,c)} \iff CDE(b) \quad \text{and} \quad FRM_{\text{free}}(c) \text{ and one of :}$$

$$CLV(c,1) = 1 \quad \text{and}$$

$$[ELN(CLV(c,2), b, d) \quad \text{and} \quad ELN(CLV(c,3), b, e)] \implies d = e \ ;$$

[This is the case where formula$\#c$ is $t \approx s$,
whose code is $< 1, \#t, \#s >$,
so $\#t = CLV(c,2)$ and $\#s = CLV(c,3)$.
The condition involving " $\implies$ " can be rewritten as
$\min\{d \mid ELN(CLV(c,2), b, d) = 1\} = \min\{e \mid ELN(CLV(c,3), b, e) = 1\}$].

$$\text{or} \quad CLV(c,1) = 2 \quad \text{and}$$

$$[ELN(CLV(c,2), b, d) \quad \text{and} \quad ELN(CLV(c,3), b, e)] \implies d < e \ ;$$

[This is the case where formula$\#c$ is $t < s$,
whose code is $< 2, \#t, \#s >$,
so $\#t = CLV(c,2)$ and $\#s = CLV(c,3)$.]

$$\text{or} \quad CLV(c,1) = 3 \quad \text{and} \quad TRU(b, CLV(c,2)) = 0 \neq 1! \ ;$$

[Here formula$\#c$ is $\neg F$, where $\#F = CLV(c,2)$.]

$$\text{or} \quad CLV(c,1) = 4 \quad \text{and} \quad TRU(b, CLV(c,2)) \quad \text{and} \quad TRU(b, CLV(c,3)) \ ;$$

[Here formula$\#c$ is $F \wedge G$, with $\#F = CLV(c,2)$ and $\#G = CLV(c,3)$.]

This shows TRU to be primitive recursive.

Without the restriction that F be quantifier-free, we would definitely have a non-recursive function here.

PLC′$_{a=0}$:

By definition,

$PLC'(c,b,0) \iff$ c is the code of a command C, and b is the code of a tuple $\vec{v} = (v_0, \cdots, v_n)$ such that, with initial bin contents $v_0, \cdots, v_n, \underline{0}$, when the command in C is "executed", the computation immediately terminates "without doing anything".

An example of such a command, for any input, is $\mathbf{whdo}(0 \approx 1)(D)$, for any command D . Alternatively, replace the condition $0 \approx 1$ by one which fails for the given input, but might be true sometimes. The number of steps in the computation is zero. Since this is pretty much the exact opposite to what is referred to as an infinite loop, such a computation might be jokingly referred to as an **infinitesmal non-loop** ! We must, of course, consider every possibility when establishing the theory. One could also have a concatenation of infinitesmal non-loops, so there are two sets of conditions in the following analysis.

Thus we have the following :

$$\underline{PLC'(c,b,0)} \qquad \iff \qquad CDE(b) \quad \text{and} \quad COM(c)$$

and one of :

$$CLV(c,1) = 3 \quad \text{and} \quad TRU(b, CLV(c,2)) = 0 \neq 1! \quad ;$$

or $CLV(c,1) = 2$ and $PLC'(CLV(c,2), b, 0)$ and $PLC'(CLV(c,3), b, 0)$.

This <u>begins</u> to show PLC' to be primitive recursive.

Exercise. Show that $\mathbf{whdo}(F)(C)$ would produce an infinite loop at any input for which F is true but C produces an infinitesmal non-loop.

[To reinforce the reader's impression that the author has an impaired sense of humour, I might add that this is one place where the product of zero and infinity must be thought of as infinity. The command causes infinitely many loops each of which should take zero time !
An example is $\mathbf{whdo}\ (0 \approx 0)\ (\mathbf{whdo}(0 \approx 1)(x_0 \dot{-} x_0))$, the silliest program in the history of the universe, but which makes a good T-shirt.]

PLC′$_{a>0}$:

By definition, for $a > 0$,

$PLC'(c, b, a) \iff$ a is the code of a marked command M whose underlying command C has code c, and b is the code of a tuple $\vec{v} = (v_0, \cdots, v_n)$ such that, when 'program' C is to be executed with initial bin contents $v_0, \cdots, v_n, \underline{0}$, the mark in M is on the first atomic sub-command to be executed.

Thus we have the following :

$$a > 0 \text{ and } PLC'(c, b, a) \qquad \Longleftrightarrow \qquad CDE(b) \quad \text{and} \quad SMC(a, c)$$

[so no need for actually saying $COM(c)$ and $MRK(a)$], and one of :

$$CLV(c, 1) = 1 \quad \text{and} \quad a = c \quad ;$$

[Here the command has the form $(x_j \dot{\leftarrow} s)$,
so not much is happening other than marking it.]

$$\text{or} \quad CLV(a, 1) = CLV(c, 1) = 2 \quad \text{and} \quad PLC'(CLV(c, 2), b, CLV(a, 2)) \quad ;$$

([Here the command has the form $(D; E)$,
and it reduces to looking within D
and finding the first atomic sub-command to be executed.]

$$\text{or} \quad CLV(a, 1) = 4 = 2CLV(c, 1) \ , \ PLC'(CLV(c, 2), b, 0) \ ,$$

$$\text{and } PLC'(CLV(c, 3), b, CLV(a, 3)) \quad ;$$

([Here the command has the form $(D; E)$,
but D on its own would produce an infinitesmal non-loop,
so it reduces to looking within E
and finding the first atomic sub-command to be executed.]

$$\text{or} \quad CLV(a, 1) = 3 \ , \ TRU(b, CLV(c, 2)) \quad \text{and} \quad PLC'(CLV(c, 3), b, CLV(a, 3)) \ .$$

[Here the command has the form $\mathbf{whdo}(F)(D)$
marked just as D would be,
since F is true at the initial bin contents .]

These last two pages show PLC' to be primitive recursive.

$\underline{\mathbf{PLC_{c=0}}}$:

By definition,

$PLC(a,b,0) \iff$ a is the code of a marked command M, and b is the code of a tuple $\vec{v} = (v_0, \cdots, v_n)$ such that, with bin contents $v_0, \cdots, v_n, \underline{0}$, after the marked atomic sub-command in M is executed, the computation terminates.

Thus we have the following :

$$\underline{PLC(a,b,0)} \qquad \Longleftrightarrow \qquad CDE(b) \quad \text{and} \quad MRK(a) \text{ and one of :}$$

$$CLV(a,1) = 1 \ ;$$

[Here the entire marked command$\#a$ has the form $\boxed{x_j \div s}$.]

$$\text{or} \quad CLV(a,1) = 2 \quad \text{and} \quad PLC(CLV(a,2),b,0) \ ; \quad \text{and} \quad PLC'(CLV(a,3),b,0) \ ;$$

[Here marked command$\#a$ has the form $(D;E)$,
with the mark inside D,
and the mark wants to jump to E,
but E then produces an infinitesmal non-loop.]

$$\text{or} \quad CLV(a,1) = 4 \quad \text{and} \quad PLC(CLV(a,3),b,0) \ ;$$

[Here marked command$\#a$ has the form $(D;E)$,
with the mark inside E,
and $CLV(a,3) = \#_{MRK}E$.]

$$\text{or } CLV(a,1) = 3 \, , \ PLC(CLV(a,3),b,0) \text{ and } TRU(b, CLV(a,2)) = 0 \neq 1!.$$

[Here marked command$\#a$ has the form $\mathbf{whdo}(F)(C)$.
So $CLV(a,3) = \#_{MRK}C$ and $CLV(a,2) = \#_{FRM}F$.
The middle condition says that the program would terminate with tuple $\#b$ in the bins if the marked command just executed were C , marked as in the larger command $\mathbf{whdo}(F)(C)$ above .
The last condition in the display says that F is false at tuple $\#b$.]

This starts to show PLC to be primitive recursive.

$\underline{\mathbf{PLC_{c>0}}}$:

By definition,

$[c > 0$ and $PLC(a,b,c)] \iff$ b is the code of a tuple $\vec{v} = (v_0, \cdots, v_n)$ and both a and c are codes of marked commands, say M and N, which have the same underlying command C, such that (while the 'program' C is being executed—but that's more-or-less irrelevant) when the marked atomic sub-command within M is executed producing bin contents $v_0, \cdots, v_n, \underline{0}$, then the next marked command to be executed is N.

Thus we have the following :

$$\underline{c > 0 \text{ and } PLC(a,b,c)} \qquad \iff \qquad CDE(b) \quad \text{and} \quad CLV(a,1) \neq 1$$

$$\text{and} \quad \exists\ d \text{ with } SMC(a,d) \text{ and } SMC(c,d) \text{ and one of :}$$

$$CLV(a,1) = 2 = CLV(c,1) \quad \text{and} \quad PLC(CLV(a,2), b, CLV(c,2)) \ ;$$

[Here C has the form $(D;E)$,
and the marks in both M and N are within D.
The condition involving "$\exists$" can be rewritten as
$\min\{\ d \mid SMC(a,d) = 1\ \} = \min\{\ d \mid SMC(c,d) = 1\ \}$].

$$\text{or} \quad CLV(a,1) = 4 = CLV(c,1) \quad \text{and} \quad PLC(CLV(a,3), b, CLV(c,3)) \ ;$$

[Here again C has the form $(D;E)$,
but the marks in both M and N are within E .]

$$\text{or} \quad CLV(a,1) = 2 \neq 4 = CLV(c,1) \quad \text{and} \quad PLC(CLV(a,2), b, 0)$$

$$\text{and} \quad [SMC(c,d) \implies PLC'(d, b, CLV(c,3))] \ ;$$

[Once again C has the form $(D;E)$, but the mark
in M is within D and 'jumps' to one in N which is within E.
Read the definitions of $PLC_{c=0}$ and PLC' carefully again to see this!
The condition involving "$\implies$" can be rewritten as
$PLC'(\min\{\ d \mid SMC(a,d) = 1\ \}, b, CLV(c,3))$.]

or $CLV(a,1) = 3 = CLV(c,1)$ and

$$\{ \text{ either (I) : } PLC(CLV(a,3), b, CLV(c,3))$$
$$\text{or (II) : } PLC(CLV(a,3), b, 0) \text{ and } TRU(b, CLV(a,2))$$
$$\text{and } [SMC(c,d) \Rightarrow PLC'(d, b, CLV(c,3))]\} \quad ;$$

[Here C has the form $\mathbf{whdo}(F)(D)$,
In case (I) above, the mark 'moves' from M to N as though the command were just D. In case (II), M is marked in a way which would terminate the execution were the command just D. But F is true at tuple$\#b$, so N now becomes marked in such a way as though the program D were just beginning its execution with tuple $\#b$ as input .]

This, combined with the case when $c = 0$ just previous, shows PLC to be primitive recursive.

Finally, we can combine all this to show that KLN_t is recursive. First define the recursive relation HST ('could be the code of a history') by

$$\underline{HST(h)} \qquad \Longleftrightarrow \qquad CDE(h) \text{ and } ODD(CLV(h,0)) \text{ and}$$

$$CDE(CLV(h, 2i+1) \text{ for } 1 \leq 2i+1 \leq CLV(h,0); \text{ and}$$
$$CLV(CLV(h,2i+1),0) = CLV(CLV(h,2j+1),0) \text{ for } 0 < i < j < CLV(h,0)/2;$$
$$\text{and } MRK(CLV(h,2i)) \text{ for } 0 < i < CLV(h,0)/2 \quad .$$

Thus $HST(h)$ holds when h has a fighting chance to be the code of the history of an **ATEN**-computation, in the sense that it is the code of a tuple of odd length, whose odd slots are themselves codes of tuples all of the same length, and whose even slots are the codes of marked commands.

This is motivated by the following. Given $\vec{v} = (v_1, \cdots, v_t)$ and a command C from **ATEN**, such that we have a *terminating* computation when the 'program' C is executed with $(0, v_1, \cdots, v_t, \underline{0})$ as initial bin contents, define a **history** of this computation to be

$$(\ \vec{v}^{(0)} \ , \ M_1 \ , \ \vec{v}^{(1)} \ , \ M_2 \ , \ \vec{v}^{(2)} \ , \cdots , \ M_\ell \ , \ \vec{v}^{(\ell)} \ ) \ \ ,$$

where :
there are "ℓ" steps in the computation ;
each of $\vec{v}^{(0)}$, $\vec{v}^{(1)}$, $\cdots$, $\vec{v}^{(\ell)}$, is an $(n{+}1)$-tuple,
for some $n \geq \max\{ \ t \ , \max\{j : x_j \text{ appears in } C\} \ \}$;
(choice of n being the only non-uniqueness)
each of $M_1, \cdots, M_\ell$ is a marked command whose underlying command is C ;
the history contains exactly all the contents of bins 0 to n as they appear during the computation [so $\vec{v}^{(0)}$ is $(0, v_1, \cdots, v_t, 0, \cdots, 0)$] ;
and exactly all the marked commands, where the mark in M_i is on the atomic sub-command of C whose execution alters the bin contents from $\vec{v}^{(i-1)}$ to $\vec{v}^{(i)}$, and where bin contents $\vec{v}^{(i)}$ are used if necessary (to check the truth of a formula in a **whdo** command) in determining the 'jump' which the mark makes, from its position in M_i to its position in M_{i+1} .

Now define the code of the above history to be

$$< \ < \ \vec{v}^{(0)} \ > \ , \ \#M_1 \ , \ < \ \vec{v}^{(1)} \ > \ , \ \#M_2 \ , \cdots , \ \#M_\ell \ , \ < \ \vec{v}^{(\ell)} \ > \ > \ .$$

It is clear that distinct histories have distinct codes.

Since we are defining $KLN_t(c, \vec{v}, h)$ to mean that c is the code of a command whose execution terminates when the input is $\vec{v}$, and that h is a corresponding history, we get the following characterization of KLN_t , *showing it indeed to be primitive recursive*, and completing the hard work of this section.

$\underline{KLN_t(c,\vec{v},h)} \quad \Longleftrightarrow \quad$ all the following hold : $COM(c)$; $HST(h)$;

$$CLV(CLV(h,1),0) > t \ \ ;$$

[Call the left-hand side $n+1$. It agrees
with $CLV(CLV(h,2k+1),0)$ for all k because $HST(h)$.]

$$CLV(CLV(h,1),i+1) = v_i \ \text{ for } \ 1 \le i \le t \ ;$$

$$CLV(CLV(h,1),i+1) = 0 \ \text{ for } \ i = 0 \ \text{ or } \ t < i \le n \ ;$$

and, with ℓ defined by $2\ell + 1 = CLV(h,0)$:

$$PLC'(c, CLV(h,1), 0) \ \text{ if } \ell = 0 \ ;$$

$$PLC'(c, CLV(h,1), CLV(h,2)) \ \text{ if } \ell > 0 \ ;$$

$$OUT(CLV(h,2k-1), CLV(h,2k), CLV(h,2k+1)) \ \text{ for } \ 1 \le k \le \ell \ ;$$

$$PLC(CLV(h,2k), CLV(h,2k+1), CLV(h,2k+2)) \ \text{ for } \ 1 \le k < \ell \ ;$$

$$\text{and } \ PLC(CLV(h,2\ell), CLV(h,2\ell+1), 0) \ \text{ if } \ell > 0 \ .$$

There is still the easy work of dealing with

$$PRINT \ : \ \mathbf{N} \ \rightarrow \ \mathbf{N} \ .$$

Define it by

$$PRINT(a) \ := \ CLV(CLV(a, CLV(a,0)), 1) \ .$$

So certainly $PRINT$ is a jolly little primitive recursive function.

Suppose that C is a command such that

$$||C||(0, v_1, \cdots, v_t, \underline{0}) \ \neq \ err \ .$$

That is, we have a non-terminating computation when starting with $v_1, \cdots, v_t$ in bins 1 to 't' , and zeroes in all other bins. Let

$$h \ = \ < \ < \ \vec{v}^{(0)} \ > \ , \ m_1 \ , \ < \ \vec{v}^{(1)} \ > \ , \ m_2 \ , \cdots , \ m_\ell \ , \ < \ \vec{v}^{(\ell)} \ > \ >$$

be the (nearly) unique number for which

$$KLN_t(\#C, v_1, \cdots, v_t, h) \ = \ 1 \quad .$$

Then $CLV(h, 0) = 2\ell + 1$, and

$$CLV(h, CLV(h, 0)) \ = \ CLV(h, 2\ell + 1) \ = \ < \ \vec{v}^{(\ell)} \ > \quad .$$

Thus

$$PRINT(h) \ = \ CLV(< \ \vec{v}^{(\ell)} \ >, 1) \ = \ \text{the first slot in} \ \ \vec{v}^{(\ell)} \quad ,$$

(which is $(\ \vec{v}^{(\ell)} \ )_0$ in our indexing). Thus $PRINT(h)$ gives the entry in bin zero at the termination of the computation above; that is, if f is the function being computed, then

$$f(v_1, \cdots, v_t) \ = \ PRINT(\min\{ \ h \ : \ KLN_t(\#C, v_1, \cdots, v_t, h) = 1 \ \}) \quad ,$$

which is exactly what is needed for Subsection IV-1 to apply.

Checking that all the relations and total functions in the present subsection are actually primitive recursive, rather than just recursive, is not especially important. That they are is not surprising, in view of the fact that it was this class of *primitive* recursive functions which Gödel invented and used in his huge discoveries related to incompleteness.

Note that we have now filled in all the details needed for the important results in Subsection IV-1.

IV-4: Primitive Recursion in ATEN/BTEN.

This section is edifying (I think!), but not essential to the main purpose, so will be relegated to smaller print. Earlier we abbreviated "is a recursive function" to "is $\mathcal{RC}$". We shall revert to that again, and also use "is $\mathcal{PRC}$" to abbreviate "is a primitive recursive function".

Rather happily, it turns out that an easily described sublanguage of **ATEN**, which we'll call **PTEN**, computes a set of functions which is exactly the set of primitive recursive functions. **PTEN** is simply the language generated by the assignment commands $x_i \Leftarrow t$ using concatenation (i.e. ";") plus a *proper* subset of **whdo** commands : those in which the condition takes the form $x_a < x_b$, where the variables x_a and x_b are unaffected by the command, except for a 'last-minute' subcommand which increases x_a by 1 (until v_a in bin number a reaches the unchanging value v_b, of course). Execution of such a while-do command is effectively just repetition of a fixed command a number of times equal to the value in some bin that is unaffected by the fixed command. In particular, no infinite loop can occur—after all, primitive recursive functions *are* total functions.

Let's be more precise about the definition : the set of commands in **PTEN** is defined inductively to be the smallest set of finite strings of symbols containing :

(i) $x_i \Leftarrow t$ for all i and all terms t;

(ii) $(C; D)$ whenever both C and D are in **PTEN**;

(iii) $\mathbf{whdo}(x_a < x_b)(C\ ;\ x_a \Leftarrow 1 + x_a)$ whenever C is in **PTEN**, as long as, for any subcommand $x_i \Leftarrow t$ of C, we have $a > i$ and $b > i$.

Theorem IV-4.1: *A function is primitive recursive if and only if there is a command in* **PTEN** *which computes it.*

Proof. To show that being primitive recursive implies the existence of a **PTEN** command, we must show that the starter functions are **PTEN**-computable, and that the set of **PTEN**-computable functions is closed under composition and primitive recursion. This is just like the proof in Section III, except that minimization is replaced by primitive recursion.

When dealing with composition in Section III, we did use the $\mathcal{BE}$-command construction (not occurring in **PTEN**). But this is easily eliminated, just as we did in Subsection IV-2, when showing that **BTEN**-computable implies **ATEN**-computable.

To deal with primitive recursion, suppose that f is given as follows, where we already have **PTEN** commands $C[g]$ and $C[h]$ which strongly compute g and h :

$$f\ :\ \begin{cases} (x_1, \cdots, x_k, 0) & \mapsto \quad h(x_1, \cdots, x_k)\ ; \\ (x_1, \cdots, x_k, n+1) & \mapsto \quad g(x_1, \cdots, x_k, n, f(x_1, \cdots, x_k, n))\ . \end{cases}$$

Choose some $N \geq 1$ so that, for $j \geq k + N$, no subcommand $x_j \Leftarrow t$ occurs in either $C[g]$ or $C[h]$. Then it is readily seen that the following command from **PTEN** computes f, as required:

$(x_{k+N+1} \leftarrow x_1$;
$x_{k+N+2} \leftarrow x_2$;
•
•
•
$x_{2k+N} \leftarrow x_k$;
$x_{2k+N+1} \leftarrow x_{k+1}$;
$C[h]$; INITIAL VALUE
$x_{k+N} \leftarrow 0$; SET COUNTER
while $x_{k+N} < x_{2k+N+1}$
do $(x_1 \leftarrow x_{k+N+1}$;
$x_2 \leftarrow x_{k+N+2}$;
•
•
•
RESTORE
$x_k \leftarrow x_{2k+N}$;
$x_{k+1} \leftarrow x_{k+N}$; VARYING 2nd LAST SLOT
$x_{k+2} \leftarrow x_0$; INDUCTIVE VALUE
$C[g]$; NEW VALUE
$x_{k+N} \leftarrow 1 + x_{k+N}))$ INCREMENT COUNTER/VARYING 2nd LAST SLOT

Note that, as required for **PTEN**, the variable x_{k+N} does not occur in a subcommand $x_{k+N} \leftarrow t$ anywhere till the end in the concatenated command within the **whdo** part, and neither does x_{2k+N+1} .

The special case when $k = 0$ needs to be modified slightly, since h is then just a number, not a function.

Conversely, we wish to show that, for any ℓ, the function of "ℓ" variables defined by a given C from **PTEN** is in fact primitive recursive. We proceed by induction to prove that *each such command C is primitive in the following sense : "the jth output of C", namely,*

$$(v_0, v_1, \cdots, v_N) \ \mapsto \ (||C||(v_0, v_1, \cdots, v_N, \underline{0}))_j$$

is a $\mathcal{PRC}$ function of "$N+1$" variables for every N and j. (In particular, we're stipulating that $||C||(\underline{v}) \neq err$ for any $\underline{v}$.

The required result is then the immediate consequence that

$$(v_1, v_2, \cdots, v_\ell) \ \mapsto \ (||C||(0, v_1, \cdots, v_\ell, \underline{0}))_0$$

is $\mathcal{PRC}$.

It is clear that the command $x_i \leftarrow t$ is primitive for all i and t.

Prop. IV-4.2: *If D and E are primitive, then so is $(D;E)$.*

Prop. IV-4.3: *If C is primitive, and the atomic subcommand $x_i \dot{\leftarrow} t$ occurs in C only for $i < m$, then the command*

$$\mathbf{whdo}(x_m < x_{m+1})(C\ ;\ x_m \dot{\leftarrow} 1 + x_m)$$

is also primitive.

For $a \geq m$ and $b \geq m$, except for $j = a, b, m, m+1$ (where the output is obviously primitive recursive), the given command in the last proposition has the same jth output as the command $\mathbf{whdo}(x_a < x_b)(C\ ;\ x_a \dot{\leftarrow} 1 + x_a)$ from the definition of the language, up to permuting bins b and $m+1$. Thus the one case $(a,b) = (m, m+1)$ from **IV-4.3** , together with **IV-4.2** , are all that is needed to complete the inductive proof.

Proof of IV-4.2: With $\vec{v} = (v_0, \cdots, v_N)$, let

$$||D||(v_0, \cdots, v_N, \underline{0}) \ = \ (f_0(\vec{v}), f_1(\vec{v}), \cdots, f_M(\vec{v}), \underline{0})$$

and

$$||E||(w_0, \cdots, w_M, \underline{0}) \ = \ (g_0(\vec{w}), g_1(\vec{w}), \cdots)$$

Then each f_i and g_j is $\mathcal{PRC}$, and the functions mapping $\vec{v}$ to $(||(D;E)||(v_0, \cdots, v_N, \underline{0}))_j$ and to $g_j(f_0(\vec{v}), f_1(\vec{v}), \cdots, f_M(\vec{v}))$ are actually the same function. By composition, the latter is $\mathcal{PRC}$, so the former is also, as required.

Proof of IV-4.3: Let $\vec{v} = (v_0, \cdots, v_{m-1})$. For $N > m+1$, the 'history' of a computation using the command in the proposition goes as follows:

$$(v_0, v_1, \cdots, v_m, \underline{v_\infty}) \ \mapsto \ (f_0(\vec{v}), \cdots, f_{m-1}(\vec{v}), 1 + v_m, \underline{v_\infty}) \ \mapsto$$

$$(f_0(f_0(\vec{v}), \cdots, f_{m-1}(\vec{v})), \cdots, f_{m-1}(f_0(\vec{v}), \cdots, f_{m-1}(\vec{v})), 2{+}v_m, \underline{v_\infty}) \ \mapsto \ \bullet\bullet\bullet$$

Each f_i is given to be a $\mathcal{PRC}$ function, since it is the ith output of C. There are "$v_{m+1} \check{-} v_m$" arrows "$\mapsto$" in total. Let $g_{N,j}$ be the function which gives the jth output of the command with history above, i.e. the function which we wish to prove primitive recursive. Since $g_{N-1,j}(\vec{v}) = g_{N,j}(\vec{v}, 0)$, it suffices to consider only N which are sufficiently large.

For $j > m$ and $N \geq j$, we have $g_{N,j}(v_0, \cdots, v_N) = v_j$. This takes care of large j.

For $j = m$ and all $N \geq m+1$, we have $g_{N,m}(v_0, \cdots, v_N) = v_{m+1}$.

For $j < m$, we have $g_{m+1,j} = g_{m+2,j} = \cdots$, so it remains to prove that $g_{m+1,j}$ is $\mathcal{PRC}$.

Now, for $0 \leq j < m$, define functions g_j of "$m+1$" variables by

$$g_j(\vec{v}, c) \ := \ g_{m+1,j}(\vec{v}, 0, c) \ .$$

Then

$$g_{m+1,j}(\vec{v}, d, e) \;=\; g_j(\vec{v}, e \dot{-} d) \;,$$

so it remains only to prove that all the g_j are $\mathcal{PRC}$.

A copy of the displayed history above is

$$(g_0(\vec{v},0), g_1(\vec{v},0), \cdots, g_{m-1}(\vec{v},0), v_m, \underline{v_\infty}) \;\mapsto\; (g_0(\vec{v},1), g_1(\vec{v},1), \cdots, g_{m-1}(\vec{v},1), 1{+}v_m, \underline{v_\infty}) \;\mapsto$$
$$(g_0(\vec{v},2), g_1(\vec{v},2), \cdots, g_{m-1}(\vec{v},2), 2 + v_m, \underline{v_\infty}) \;\mapsto\; \bullet\ \bullet\ \bullet$$

So we get the following system of recursions for $g_0, \cdots, g_{m-1}$:

$$g_j(\vec{v}, 0) \;=\; v_j \;\;;$$
$$g_j(\vec{v}, c+1) \;=\; f_j(g_0(\vec{v},c), g_1(\vec{v},c), \cdots, g_{m-1}(\vec{v},c)) \;.$$

A standard exercise in recursive theory now shows that each g_j is primitive recursive, completing the proof, as follows.

Define the function h of "$m+1$" variables by, for values $c \geq 0$ and $0 \leq j < m$,

$$h(\vec{v}, mc + j) \;:=\; g_j(\vec{v}, c) \qquad\qquad (**)$$

Define the function H of "$3m$" variables by

$$H(\vec{v}, d, x_1, \cdots, x_{2m-1}) \;:=\; \begin{cases} f_0(x_m, x_{m-1}, \cdots, x_1) & \text{for} \quad d \equiv 0 (\text{mod} m) \;; \\ \quad\bullet & \\ \quad\bullet & \\ f_j(x_{m+j}, \;\cdots\;, x_{1+j}) & \text{for} \quad d \equiv j (\text{mod} m) \;; \\ \quad\bullet & \\ \quad\bullet & \\ f_{m-1}(x_{2m-1}, \cdots, x_m) & \text{for} \quad d \equiv m-1 (\text{mod} m) \;. \end{cases}$$

Since each f_j is $\mathcal{PRC}$, so is H, using definition by cases.

But now, for $d \geq m$, as verified below,

$$h(\vec{v}, d) \;=\; H(\vec{v}, d, h(\vec{v}, d-1), h(\vec{v}, d-2), \cdots, h(\vec{v}, d-(2m-1))) \qquad (*)$$

(For $m \leq d < 2m-1$, there is no problem with $d-j$ negative here, in view of the definition of H.)

Combined with the initial conditions

$$h(\vec{v}, j) = g_j(\vec{v}, 0) = v_j \quad \text{for } 0 \leq j < m \;,$$

we see from $(*)$ that h is determined by a primitive recursion of depth "$2m-1$", so h is $\mathcal{PRC}$. By $(**)$, each g_j is $\mathcal{PRC}$, as required.

To verify $(*)$, and complete the proof of **IV-4.1**, write $d = mc + j$ with $c \geq 1$ and $0 \leq j < m$. Then the right-hand side of $(*)$ is, since $d \equiv j (\text{mod } m)$,

$$f_j(h(\vec{v}, d-(m+j)), h(\vec{v}, d-(m+j-1)), \cdots, h(\vec{v}, d-(1+j))) =$$
$$f_j(h(\vec{v}, m(c-1)), h(\vec{v}, m(c-1)+1), \cdots, h(\vec{v}, m(c-1)+m-1)) =$$
$$f_j(g_0(\vec{v}, c-1), g_1(\vec{v}, c-1), \cdots, g_{m-1}(\vec{v}, c-1)) \;=\; g_j(\vec{v}, c) \;=\; h(\vec{v}, d) \;.$$

The existence of **PTEN** gives us a 'self-contained' theory, not of *all* total computable functions, but rather of those which are primitive recursive. So the 'diagonize-out' argument showing TOT non-computable should have an analogue here. What it does is to give an easy proof of the existence of a total recursive function which is not primitive recursive, as we now explain.

Let $KLNP_t$ be the Kleene relation for "t"-variable functions obtained using only the history of **PTEN** computations. So

$$KLNP_t(c, \vec{v}, h) \iff KLN_t(c, \vec{v}, h) \text{ and } PTN(c)$$

where $PTN(c)$ holds if and only if c is the command number of a command in **PTEN**. We shall leave it as an exercise, done by the same tedious coding analysis as in the previous subsection, to show that PTN is recursive. Thus $KLNP_t$ is recursive also.

Now, mimicking the proof of Theorem **IV-1.7**, define

$$L(a, b) := PRINT(\min\{h : KLNP_1(b, a, h) = 1\})$$

This is clearly recursive, and is total because for every (a, b) there is an h for which $KLNP_1(b, a, h) = 1$. This is simply because every computation using a command from **PTEN** does terminate. Therefore we get a total recursive function D by defining

$$D(b) := 1 + L(b, b) .$$

Now, for every primitive recursive f of one variable, there is a b such that, for all a, we have $f(a) = L(a, b)$. This just says that $b \mapsto L(-, b)$ gives an effective list (with many repetitions) of all primitive recursive functions of one variable. That's immediate from the definition of $KLNP$, and the main theorem, **IV-4.1**, of this subsection.

Now suppose, for a contradiction, that D is primitive recursive. Choose d such that, for all a, we have $D(a) = L(a, d)$. Then we immediately get a contradiction:

$$L(d, d) = D(d) = 1 + L(d, d) .$$

Thus D is indeed a total, non-primitive recursive function.

Note carefully that the above is very far from giving an algorithm to decide whether or not a given **BTEN** command computes a primitive recursive function of, say, one variable. Subsection **IV-6** considers several such problems, ones of perhaps more interest to computer scientists than this one.

Possibly the reader would enjoy the exercise of extending this result to produce, for any $\ell \geq 1$, a total, non-primitive recursive function of "ℓ" variables. This may be done either by using the result above for $\ell = 1$, or also more directly by imitating its proof, in each case using Cantor's bijections.

Another, quite specific, function which is computable but not primitive recursive, is Ackermann's function, dealt with in IV-10. There the argument to establish non-primitivity is harder than here.

IV-5: Recursive enumerability.

Here is the main theorem, concerning functions of *one* variable, though easily generalizable to more. Remember that "*image of* " means "set of values of " (sometimes also called "range of ", though "range" is also a name used for *codomain*, always $\mathbf{N}$ here).

Theorem IV-5.1. *Let D be a non-empty set of natural numbers. Then the following are equivalent :*
(i) D is the domain of some $\mathcal{RC}$ function.
(ii) D is the image of some $\mathcal{PRC}$ function.
(iii) D is the image of a total $\mathcal{RC}$ function.
(iv) D is the image of some $\mathcal{RC}$ function.

Corollary IV-5.2. *Actually, (i) and (iv) are equivalent for* $\mathsf{all}\ D \subset \mathbf{N}$.

This is clear since the function with empty domain is certainly recursive, and has empty image.

Definition. Sets D as above (including $\emptyset$) are called *recursively enumerable* or *listable*. We shall shorten this to $\mathcal{RE}$.

Condition (iii) is the motivation for this terminology. By running a command for the function successively on inputs $0, 1, 2, \cdots$, we get a mechanism (as discussed intuitively very early in this work) which lists the values of the function, the members of D.

As exercises extending the above theorem, prove that : the graph of a function $\mathbf{N} \to \mathbf{N}$ is $\mathcal{RE}$ (vis-a-vis Cantor's bijection from $\mathbf{N}^2$ to $\mathbf{N}$) iff the function is (partial) recursive; and that an infinite set is $\mathcal{RE}$ iff it is the image of a *injective* total recursive function. This latter exercise makes the term "recursively enumerable" even more sensible, since now the listing of values has no repeats in it.

Definition. A set $D \subset \mathbf{N}$ is called *recursive* (or $\mathcal{R}$, or *decidable*) if and only if its *characteristic* function is recursive. That function is the predicate (or relation)

$$\chi_D \ : \ c \ \mapsto \ \begin{cases} 1 & \text{if } c \in D \ ; \\ 0 & \text{if } c \notin D \ . \end{cases}$$

Theorem IV-5.3. *The set D is $\mathcal{R} \iff$ both D and its complement, $\mathbf{N} \setminus D$, are $\mathcal{RE}$.*

Another exercise is to show that the recursive infinite sets are exactly the images of *increasing* functions $\mathbf{N} \to \mathbf{N}$ which are recursive.

The two theorems above will be proved below after some comments and definitions.

The existence of non-recursive sets was more-or-less discovered in the 19th century, by cardinality considerations, even before the definition was clear. The major discovery in the 20th century was of $\mathcal{RE}$ sets which are not recursive. Examples are the sets of code numbers: of derivable formulae in many $1^{\underline{\text{st}}}$order theories, of commands which compute

functions which halt with that code number as input, etc. See the next subsection for lots more. See the brief discussion of Hilbert's 10th problem in Section XXXXX for the most famous example related to mainstream classical mathematics.

The cardinality argument above extends to an indirect proof of the existence of (huge numbers of!) non-$\mathcal{RE}$ sets. One observes that the set of all $\mathcal{RE}$ sets is *countable*, whereas Cantor famously showed that the set of *all* subsets of $\mathbf{N}$ is *un*countable.

Specific examples of sets which are not even $\mathcal{RE}$ are the sets of code numbers: of formulae in $1^{\underline{\text{st}}}$order number theory which are true in $\mathbf{N}$ (see Appendix L of [**LM**]); and of **BTEN** commands which compute total functions (see Theorem **IV-1.7**).

To prove these theorems, we need a few more technicalities, extending the menagerie of recursive functions from Subsection IV-3.

Definitions. For $t \geq 1$ and $k \geq 0$, define

$$BIN_t(c, v_1, \cdots, v_t, 2k) \ := \ CLV(\min\{h \ : \ KLN_t(c, v_1, \cdots, v_t, h) = 1\}, 2k) \ ; \qquad (*)$$

$$MCM_t(c, v_1, \cdots, v_t, 2k+1) \ := \ CLV(\min\{h \ : \ KLN_t(c, v_1, \cdots, v_t, h) = 1\}, 2k+1) \ . \qquad (*)$$

Thus, an h (if it exists) for which $KLN_t(c, \vec{v}, h) = 1$ has the form

$$h \ = \ < BIN_t(c, \vec{v}, 0), MCM_t(c, \vec{v}, 1), BIN_t(c, \vec{v}, 2), \cdots, MCM_t(c, \vec{v}, 2\ell-1), BIN_t(c, \vec{v}, 2\ell) > \ . \ (**)$$

"BIN" and "MCM" stand for *bin contents* and *marked command* respectively.

As defined above, these functions are partial, and undefined on $(c, \vec{v}, n)$ if n has the wrong parity, if c is not the code of a command, or c is the code of a command which, when applied to $\vec{v}$ as input, gives a non-terminating computation. The latter condition can be eliminated by going right back to the specifications which showed that KLN is recursive. The infinite sequence, corresponding to the finite sequence within the $<\quad>$ in the last display, would give the entire 'history' of a non-terminating computation. We shall need this extension of the functions just below.

So the following equalities redefine BIN and MCM more generally, and show them to be recursive : Given $c \in \mathbf{N}$ and $\vec{v} \in \mathbf{N}^t$, let

$$n \ := \ \max\{ \ t \ , \max\{j : x_j \text{ appears in command\#}c\} \ \} \ .$$

$$BIN_t(c, \vec{v}, 0) \ := \ < 0, \vec{v}, 0, \cdots, 0 >$$

where the right-hand side is the code of an $(n+1)$-tuple;

$$MCM_t(c, \vec{v}, 1) \ := \ \min\{ \ \ell \ : \ PLC'(c \ , \ BIN_t(c, \vec{v}, 0) \ , \ \ell) \ \} \ ;$$

$$BIN_t(c, \vec{v}, 2k) \ := \ \min\{ \ \ell \ : \ OUT(BIN_t(c, \vec{v}, 2k-2) \ , \ MCM_t(c, \vec{v}, 2k-1) \ , \ \ell) \ \} \ ;$$

$$MCM_t(c, \vec{v}, 2k+1) \ := \ \min\{ \ \ell \ : \ PLC(MCM_t(c, \vec{v}, 2k-1) \ , \ BIN_t(c, \vec{v}, 2k) \ , \ \ell) \ \} \ .$$

Possibly $(*)$ (not used later!) fails for this redefinition. But $(**)$ certainly holds.

Now define a relation $STEP_t$ by

$$STEP_t(c, v_1, \cdots, v_t, s) \iff \exists h \text{ with } CLV(h, 0) \leq 2s+1 \text{ and } KLN_t(c, v_1, \cdots, v_t, h) = 1 \ .$$

That is, $\cdots\cdots \iff c$ is the code of a command which, with input $(v_1, \cdots, v_t)$, produces a computation that terminates in at most "s" steps.
Clearly, $\cdots\cdots \iff \exists k \leq s$ with $PLC(MCM(c, v_1, \cdots, v_t, 2k-1), BIN(c, v_1, \cdots, v_t, 2k), 0)$.

The latter shows $STEP_t$ to be recursive. (It sure wouldn't be without the "$\leq s$".) To see recursivity, one first computes $COM(c)$. When it's 0, so is $STEP_t$. Otherwise, MCM and BIN in the last characterization above can be computed because the parities of their last arguments are correct. And there are only finitely many PLC's to compute to check the condition.

It is convenient to extend the language by allowing assignment commands which insert values of "known-to-be" total recursive functions into bins. Let f be such a function of "ℓ" variables, let $C[f]$ be a command which strongly computes it. Let $t_1, \cdots, t_\ell$ be terms in $1^{\underline{\text{st}}}$order number theory. Define $x_i \Leftarrow f(t_1, \cdots, t_\ell)$ to be an abbreviation for the following command, where we use any sufficiently large natural number N, larger than ℓ, i and all subscripts on variables appearing in $C[f]$ or in any t_j :

$$\begin{aligned}\mathcal{B}_{N,N+1,\cdots,2N+\ell}(&x_N \Leftarrow x_0 \ ; \ x_{N+1} \Leftarrow x_1 \ ; \ \cdots \ ; \ x_{2N-1} \Leftarrow x_{N-1} \ ; \\ &x_{2N+1} \Leftarrow t_1 \ ; \ \cdots \ ; \ x_{2N+\ell} \Leftarrow t_\ell \ ; \\ &x_1 \Leftarrow x_{2N+1} \ ; \ \cdots ; x_\ell \Leftarrow x_{2N+\ell} \ ; \\ &C[f] \ ; \ x_i \Leftarrow x_0 \ ; \\ &x_0 \Leftarrow x_N \ ; \ \cdots \ ; \ x_{i-1} \Leftarrow x_{N+i-1} \ ; \\ &x_{i+1} \Leftarrow x_{N+i+1} \ ; \ \cdots \ ; \ x_{N-1} \Leftarrow x_{2N-1})\mathcal{E}\end{aligned}$$

Another convenient extension is to have an arbitrary recursive relation R appear as the condition in **whdo** and **ite** commands. The latter is reducible to the former by the theorem that $\mathcal{BC} \Rightarrow \mathcal{AC}$. As for the former, let D be a **BTEN** command, let R be a (total) recursive relation defined on $\mathbf{N}^\ell$, let $C[R]$ be a command which strongly computes it, let $t_1, \cdots, t_\ell$ be terms in $1^{\underline{\text{st}}}$order number theory, and choose $N > \ell$ so that x_j appears in $C[R]$ or D or any t_i only for $j < N$. Define

while $R(t_1, \cdots, t_\ell)$
do D

to be an abbreviation for the following command :

$$\begin{aligned}&\mathcal{B}_N(x_N \Leftarrow x_0 \ ; \ x_0 \Leftarrow R(t_1, \cdots, t_\ell) \ ; \\ &\quad \textbf{while } x_0 \approx 1 \\ &\qquad \textbf{do } (x_0 \Leftarrow x_N \ ; \ D \ ; \ x_N \Leftarrow x_0 \ ; x_0 \Leftarrow R(t_1, \cdots, t_\ell)) \\ &\quad x_0 \Leftarrow x_N \ )\mathcal{E}\end{aligned}$$

When either of these abbreviations is used, it is assumed that N is chosen large enough that the semantic effect of the command (of which the abbreviation is a part) becomes what is desired. More explicit notation would fasten this down quite rigorously, but is unnecessary for the limited use to which this is put here. But if doing a completely thorough general study, one ought to give notation so that the semantic effect of the abbreviation is uniquely specifiable in terms of the explicit notation for the abbreviation.

Proof that (i) implies (ii) and (iii) in IV-5.1. Let $\emptyset \neq D \subset \mathbf{N}$ and let $g : D \to \mathbf{N}$ be an $\mathcal{RC}$ function. Suppose that $C[g]$ strongly computes g. Then the command following the next paragraph computes a total function f such that f maps $\mathbf{N}$ onto exactly the subset D. After writing out the command we'll explain this.

This establishes (i) $\Rightarrow$ (iii). We shall leave it as an exercise for the reader to expand this into a purely **ATEN** command, using the definition just before this proof and the methods of Subsection IV-2. And then to note that it's actually in **PTEN**, so f is primitive, proving (ii). In any case (ii) $\Rightarrow$ (iii) is trivial, and so is (iii) $\Rightarrow$ (iv), so only proving that (iv) $\Rightarrow$ (i) will remain to finish **IV-5.1**.

$(x_5 \leftarrow 1 + x_1$;
$x_3 \leftarrow 0$; $x_1 \leftarrow NAC_1(0)$; $x_2 \leftarrow NAC_2(0)$;
$x_4 \leftarrow 0$;
while $x_4 < x_5$
do **if** $STEP_1(\text{“}\#C[g]\text{”}, x_1, x_2)$
thendo $(x_0 \leftarrow x_1$
$x_4 \leftarrow 1 + x_4$
$x_3 \leftarrow 1 + x_3$; $x_1 \leftarrow NAC_1(x_3)$; $x_2 \leftarrow NAC_2(x_3)$ $)$
elsedo $(x_3 \leftarrow 1 + x_3$; $x_1 \leftarrow NAC_1(x_3)$; $x_2 \leftarrow NAC_2(x_3)$ $)$ $)$

The ‘syntactic version’, “v”, of a natural number v is simply a suitably bracketted string a 1's and +'s. See the last part of Appendix A. Using it above in the command, we are being fussy about the distinction between syntax and semantics (in this case, between a constant term in the language of number theory and an actual number). Just below, we have very loose morals in that respect!

Explanation : Given c in bin 1, we want to systematically look at all pairs of natural numbers (x_1, x_2), checking each time whether the command $C[g]$ with input x_1 terminates in at most “x_2” steps. There is no problem with those x_1 where it doesn't terminate at all, since we simply stop trying, once it hasn't terminated after “x_2” steps. Once we have come to $c^{\underline{\text{th}}}$ time such a pair has been found, we put the first slot of the pair, x_1, in bin 0, and terminate this computation. It is clear from this that every x_1 in the domain of g ends up in the image of this function f just computed (infinitely often!); that only such numbers are in that image; and that this computation does terminate for every input c (i.e. f is total—there will be a $c^{\underline{\text{th}}}$ time as above, even if only one number exists in the domain of g). The systematic journey through all pairs is a trip through the “Cantor diagram”, the picture one uses to define the famous bijection $CAN : \mathbf{N} \times \mathbf{N} \to \mathbf{N}$. The components of its inverse are denoted NAC_1 and NAC_2 .
As for the lines of the above command :
Line 1 stores the input's successor safely in bin 5, never later to be interfered with.
Line 2 starts the counter x_3, and starts the trip at $(0, 0)$ in the bottom left corner of the Cantor diagram. We could have just written 0 instead of $NAC_i(0)$.
Line 3 starts the counter x_4 which counts the number of times a pair has been found as above.
Line 4 starts the loop, which is executed until that number of pairs exceeds the input.

Thus the command above does what we said it would. Notice that it is a rigorous version of the very informal argument right at the beginning of the work which explained how to produce a mechanism which lists the domain of the partial function which another given mechanism computes.

Proof that (iv) implies (i) in IV-5.1. Let $D \subset \mathbf{N}$ and let $f : D \to \mathbf{N}$ be an $\mathcal{RC}$ function. Suppose that $C[f]$ strongly computes f. Then the command below computes the function g such that

$$g(v) \ : \ = \ \begin{cases} 0 & \text{if } v \in f(D) \ ; \\ \text{undefined} & \text{otherwise .} \end{cases}$$

Since the domain of g is the image of f, we'll be finished.

$(x_0 \leftarrow 1 + x_1 \ ;$
$x_4 \leftarrow x_1 \ ;$
$x_3 \leftarrow 0 \ ; \ x_1 \leftarrow NAC_1(0) \ ; \ x_2 \leftarrow NAC_2(0) \ \ ;$
while $\neg x_4 \approx x_0$
do **if** $STEP_1(\text{“}\#C[f]\text{”}, x_1, x_2)$
thendo $(\mathcal{B}_{1,2,3,4}C[f]\mathcal{E} \ ;$
$x_3 \leftarrow 1 + x_3 \ ; \ x_1 \leftarrow NAC_1(x_3) \ ; \ x_2 \leftarrow NAC_2(x_3) \ \ )$
elsedo $(x_3 \leftarrow 1 + x_3 \ ; \ x_1 \leftarrow NAC_1(x_3) \ ; \ x_2 \leftarrow NAC_2(x_3) \ \ ) \ )$
$x_0 \leftarrow 0 \ )$

Explanation : Given c in bin 1, we want to systematically look at all pairs of natural numbers (x_1, x_2), checking each time whether the command $C[f]$ with input x_1 terminates in at most "x_2" steps. There is no problem with those x_1 where it doesn't terminate at all, since we simply stop trying, once it hasn't terminated after "x_2" steps. Each time a pair is found, the value of f on x_1 is compared to the input c. Once such a pair where they agree is found, bin 0 is given the number 0, and the computation is terminated. If no such pair is found, termination never occurs.
As for the lines of the above command :
Line 1 stores the input's successor in bin zero, to force at least one loop below to happen.
Line 2 stores the input safely in bin 4, never later to be interfered with.
Line 3 starts the counter x_3, and starts the trip through the Cantor diagram at $(0, 0)$, its bottom left corner. We could have just written 0 instead of $NAC_i(0)$.
Line 4 starts the loop, which is executed until a value of f is found which agrees with bin 4's content.
Line 9 at the bottom inserts g's only value, namely 0, in bin zero, but this line won't be reached if f never takes the value c.

Thus the command above does what we said it would. Notice that it is a rigorous version of the very informal argument right at the beginning of the work which explained how to produce a mechanism which computes a partial function that is defined on precisely the numbers which another given mechanism lists.

Proof of Theorem IV-5.3. Suppose that D is $\mathcal{R}$ and command C decides membership in D; that is, it strongly computes the characteristic function. Let B be the command $\mathbf{whdo}(0 \approx x_0)(\mathcal{B}_1 C \mathcal{E})$. It is rather easy to see that B computes the partial function with domain D which maps everything to 1. Thus D is $\mathcal{RE}$, by Theorem **IV-5.1** (i).

So, D is $\mathcal{R} \implies D$ is $\mathcal{RE}$.

Thus, D is $\mathcal{R} \implies \mathbf{N}\backslash D$ is $\mathcal{R} \implies \mathbf{N} \setminus D$ is $\mathcal{RE}$.

Conversely, suppose that both D and $\mathbf{N}\backslash D$ are $\mathcal{RE}$. By Theorem **IV-5.1** (i), choose functions f and g with domains D and $\mathbf{N}\backslash D$ respectively, and commands $C[f]$ and $C[g]$ respectively, which compute them. Consider the following command, which we claim computes the characteristic function of D, as required to prove that D is decidable :

$(x_0 \leftarrow 1 + 1$;
$x_2 \leftarrow 0$;
while $x_0 \approx 1 + 1$
do **if** $STEP_1(\text{“}\#C[f]\text{”}, x_1, x_2)$
thendo $x_0 \leftarrow 1$
elsedo **if** $STEP_1(\text{“}\#C[g]\text{”}, x_1, x_2)$
thendo $x_0 \leftarrow 0$
elsedo $x_2 \leftarrow 1 + x_2$))

Explanation : Basically we are just alternately checking f and g to see whether a given number is in the domain, once for each possible number of steps needed to compute it. Given an input c, the command above checks, for each possible number of steps, which (if either) of $C[f]$ or $C[g]$ terminates after that number of steps. At most one of them will. The counter x_2 keeps track of the number of steps referred to above. Bins number 0 and 1 remain as $(2, c, \cdots)$ through all steps, after the first step until the last step, of a computation with the above command. If c is in D, the second **if** condition always fails, and the counter x_2 in the last line eventually reaches a value where the first **if** condition holds. Then 1 is inserted in bin zero and the computation terminates. On the other hand, if c is in $\mathbf{N} \setminus D$, the first **if** condition always fails, and we end up with 0 in bin zero at termination.

So the command does indeed show D to be a recursive set.

There is another rather fundamental connection between computability and recursive enumerability. This essentially does an exercise mentioned near the beginning of this subsection. But we need to talk about subsets of $\mathbf{N}^t$ for $t > 1$ being $\mathcal{RE}$. Just below we shall take this to mean that the set is precisely the domain of some $\mathcal{BC}$-function. So the exercise can be changed to challenging you to prove that a set is recursively enumerable in this sense if and only if Cantor's bijection $\mathbf{N}^t \to \mathbf{N}$ maps it onto an $\mathcal{RE}$-subset of $\mathbf{N}$.

Theorem IV-5.4. *A function is computable if and only if its graph is recursively enumerable.*

Recall that the graph of $f : D \to \mathbf{N}$, where $D \subset \mathbf{N}^k$, is the set

$$G_f \ =: \ \{ \ (\vec{v}, f(\vec{v})) \in \mathbf{N}^{k+1} \mid \vec{v} \in D \ \} \ .$$

Here are a couple of interesting examples, sticking to $k = 2$ for simplicity only.

The universal function $U = U_1$, which is of course computable, yields a graph whose recursive enumerability corresponds to the computability of the 'partial characteristic function'

$$K''(c, v, b) = \begin{cases} 1 & \text{when command\#}c \text{ with input } v \text{ produces output } b \ ; \\ err & \text{otherwise} \quad . \end{cases}$$

Recall that the actual characteristic function is K' from the discussion just before proving Turing's theorem on the halting problem. There we showed why K' is not computable, so the graph of U is not a recursive set. So this is another example of a recursively enumerable, but undecidable set.

On the other hand, we can get a second example in a very similar manner. But instead of using U, which maps (code number of command, input number) to the *corresponding output number*, we could consider the function which maps that pair to the *code number of the history of the computation*. Then the characteristic function of the graph is none other than the Kleene computation relation. So in this case, the graph is not just recursively enumerable, it is decidable ('primitively').

Rather than proving the theorem rigorously, we'll leave that as an exercise in writing a couple of **BTEN** programs, perhaps based on the following intuitive explanations of the two halves of the theorem.

First suppose that f is computable. To get a procedure which terminates exactly when its input comes from the graph, G_f, suppose given as input $(\vec{v}, \ell) \in \mathbf{N}^{k+1}$. Run a procedure for computing f, using input $\vec{v}$. If this fails to terminate, that's fine. If it does terminate, check whether its output, i.e. $f(\vec{v})$, is the same as ℓ. If so, output 1 and stop. But if not, add in a little procedure which always produces an infinite loop.

Conversely, assume that G_f is recursively enumerable. We want a procedure for computing f itself. Fix a procedure (as produced in the above paragraph) which terminates on any input from the graph, but loops on any other input from $\mathbf{N}^{k+1}$. Get the procedure for f as follows, taking $\vec{v}$ as a typical input. Successively run steps of the earlier procedure as follows:

1st step on $(\vec{v}, 0)$; then

1st step on $(\vec{v}, 1)$; 2nd step on $(\vec{v}, 0)$; then

1st step on $(\vec{v}, 2)$; 2nd step on $(\vec{v}, 1)$; 3rd step on $(\vec{v}, 0)$; then ... etc. ...

If and when one of these steps is the last step, on, say, $(\vec{v}, \ell)$, then necessarily, $\ell = f(\vec{v})$, so stop and output ℓ. If that never happens, then $\vec{v} \notin D$, and the procedure runs without terminating, as required.

IV-6: More Undecidability Results.

This is also a sideline to our main purpose here.

For a fixed $v_0 \in \mathbf{N}$, define a relation

$$HLT_{v_0}(c) \ := \ \begin{cases} 1 & \text{if } COM(c) \text{ and } \exists h \text{ with } KLN_1(c, v_0, h) = 1 \ ; \\ 0 & \text{otherwise .} \end{cases}$$

Define

$$NTH_1(c) \ := \ \begin{cases} 1 & \text{if } COM(c) \text{ and } \nexists (v, h) \text{ with } KLN_1(c, v, h) = 1 \ ; \\ 0 & \text{otherwise .} \end{cases}$$

Define

$EQV_1(c, d) \iff COM(c), COM(d)$ and the one variable functions

defined by the corresponding commands are the same.

The condition on the right-hand side is

$$\forall v \in \mathbf{N} \ , \ ||C||(0, v, \underline{0}) = ||D||(0, v, \underline{0}) \text{ where } \#C = c \text{ and } \#D = d \ .$$

We shall prove

$$HAL_1 \text{ not } \mathcal{RC} \implies HLT_{v_0} \text{ not } \mathcal{RC} \implies NTH_1 \text{ not } \mathcal{RC} \implies EQV_1 \text{ not } \mathcal{RC} \ .$$

Since HAL_1 is indeed not a recursive function by Turing's halting problem result (Theorem **IV-1.5**), this shows that there can be no algorithms as follows :

for any choice of fixed input, to decide whether or not commands do "infinite loops" on that fixed input;

for commands, to decide whether they always do an "infinite loop", no matter what the input (this is the same as asking whether the function computed by the command has empty domain);

for pairs of commands, to decide whether the functions computed by the commands are the same (including, of course, having the same domain).

Strictly speaking, with the technical definitions above, in each case we are referring to single numbers as input, and so, functions of one variable. However, it all extends easily to any number of variables.

Proof that HLT_{v_0} is not $\mathcal{RC}$.

For fixed v_0 and v in **N**, and a command C whose code is c, define c_v to be the code of the following command D in **ATEN** :

$(x_{k+1} \Leftarrow x_1$;
while $x_1 \approx$ "v_0"
do ($x_1 \Leftarrow$ "v" ;
C ;
$x_1 \Leftarrow$ "v_0" $+ 1$) ;
$x_1 \Leftarrow x_{k+1}$;
while $\neg x_1 \approx$ "v_0"
do C

The 'syntactic version', "v", of a natural number v is simply a suitably bracketted string a 1's and +'s. See the Busy Beaver addendum to Section II. By working out in detail the code number, $\#D$, one sees that, for fixed v_0, but 'varying' v, the function $(c, v) \mapsto c_v$ is a recursive function. Note that D has the same effect

as the **BTEN** command : **ite** $(\neg x_1 \approx$ "v_0")$(C)(x_1 \Leftarrow$ "v"$; C)$.

Now define

$$G \ : \ (c, v) \ \mapsto \ \begin{cases} 0 & \text{if } COM(c) = 0 \ ; \\ HLT_{v_0}(c_v) & \text{if } COM(c) = 1 \ . \end{cases}$$

For a contradiction, assume that HLT_{v_0} is $\mathcal{RC}$. Then G is $\mathcal{RC}$. But

$$G(c, v) = 1 \iff [COM(c) \text{ and } HLT_{v_0}(c_v)] \iff [COM(c) \text{ and } ||D||(0, v_0, \underline{0}) \neq err] \iff$$

$$[COM(c) \text{ and } ||C||(0, v, \underline{0}) \neq err] \iff HAL_1(c, v) = 1 \ .$$

This shows that $G = HAL_1$. Thus HAL_1 is $\mathcal{RC}$, a contradiction .

Proof that NTH_1 is not $\mathcal{RC}$.

Define

$$F(c) \iff COM(c) \text{ and } \neg NTH_1(\#(x_1 \Leftarrow 0 \ ; \ \text{command}\#c)) \ .$$

That is, to (unnecessarily?) write it explicitly in terms of Cantor's function :

$$F(c) := COM(c)(1 - NTH_1(CTR(2, CTR(1, CTR(0, 1, 2), CTR(0, 0, 0)), c))) \ .$$

So F would be $\mathcal{RC}$ if NTH_1 were, which we assume for a contradiction. But

$$F(c) = 1 \iff [COM(c) \text{ and } \exists(v, h) \text{ with } KLN_1(\#(x_1 \Leftarrow 0 \, ; \, \text{command}\#c), v, h) = 1] \iff$$

$$[COM(c) \text{ and } \exists h \text{ with } KLN_1(c, 0, h) = 1] \iff HLT_0(c) = 1 \ .$$

This shows that $F = HLT_0$. Thus HLT_0 is $\mathcal{RC}$, a contradiction to the previous result.

Proof that EQV_1 is not $\mathcal{RC}$.

Let

$$c_0 \;=\; \#\mathbf{whdo}(0 \approx 0)(x_0 \leftarrow x_0\ )$$

Then

$$NTH_1(c) = 1 \iff [COM(c) \text{ and } \forall v \in \mathbf{N}\ ,\ ||\text{command}\#c||(0, v, \underline{0}) = err] \iff$$

$$[COM(c) \text{ and } \forall v \in \mathbf{N}\ ,\ ||\text{command}\#c||(0, v, \underline{0}) = ||\text{command}\#c_0||(0, v, \underline{0})] \iff$$

$$EQV_1(c, c_0) = 1\ .$$

But if EQV_1 were recursive, then so would be the one variable function $c \mapsto EQV_1(c, c_0)$, which coincides with NTH_1, as we've just seen. And so, EQV_1 cannot be recursive.

It is getting to the point where one might believe that almost no interesting predicate involving the code numbers of commands to be recursive. In fact, there is an amazingly general theorem along these lines. We shall stick to talking about one variable functions below. However, there are also versions of Rice's theorem for any larger fixed number of variables, for any number of variables simultaneously, and for strings in general.

Consider subsets of $COM^{-1}(1)$, the set of all code numbers of commands. Such a subset $\mathcal{S}$ will be called a *1-index set* when, for any commands C and D which compute the *same* function of one variable, either both $\#C$ and $\#D$ are in $\mathcal{S}$ or neither are.

Basically we want to consider sets of commands where membership in the set depends only on the semantic effect of the command, not on every last detail of its syntax. And then ask about decidability of the set.

For example, the following are all 1-index sets:

$\mathcal{S}_1$ = codes of commands which compute primitive recursive functions of one variable.

$\mathcal{S}_2$ = codes of commands which compute 1-variable functions which are total.

$\mathcal{S}_3$ = codes of commands which compute 1-variable functions with empty domains.

$\mathcal{S}_4$ = codes of commands which compute 1-variable functions for which some fixed v_0 is in their domains.

Applied to the latter two, the theorem below reproves two of the three results above. Applied to the first example, it shows that there is no algorithm for deciding whether a given command computes a primitive recursive function. Applied to the second, it reproves Theorem **IV-1.6**—but the earlier direct proof of **IV-1.7** is desirable because :
(i) the result is stronger, and (ii) it exhibits the difference between the informal argument at the beginning of this work, and its mathematical analogue.

Rice's Theorem IV-6.1: *Except for $\emptyset$ and $COM^{-1}(1)$ itself, no 1-index set is decidable.*

Proof: For a contradiction, assume that $\mathcal{S}$ is a 1-index non-empty proper subset of command code numbers which *is* decidable. Below we shall fix a number c_0 as indicated there, depending on two possible cases for $\mathcal{S}$.

For fixed $c_0 \in COM^{-1}(1)$, let C_0 be the command with code c_0. Now for any ("variable") $c \in COM^{-1}(1)$, with command C having code c, let N be larger than any variable's subscript appearing in C, and let D_c be the following command:

$$(\ x_N \Leftarrow x_1 \ ; \ x_1 \Leftarrow \text{“}c\text{”} \ ; \ C \ ; \ x_1 \Leftarrow x_N \ ; \ C_0 \ )$$

Directly computing the code of this command, we see that $c \mapsto \#D_c$ is recursive, and so the following function is also recursive, by the decidability of $\mathcal{S}$:

$$(*) \qquad\qquad c \ \mapsto \ \begin{cases} 1 & \text{if } \#D_c \in \mathcal{S} \ ; \\ 0 & \text{otherwise .} \end{cases}$$

Let $\mathcal{T}$ be the 1-index set of all codes c of commands C such that $||C||(0, c, \underline{0}) \neq err$. Thus the undecidability of $\mathcal{T}$ is exactly the non-recursivity of $c \mapsto HAL_1(c, c)$, which we proved in **IV-1.5**, Turing's big theorem!

[Oddly enough, despite being the most important instance of a non-decidable set of codes of commands, $\mathcal{T}$ is *not* an example of a 1-index set. In the next section, we find a command C with code c such that C computes the 1-variable function $\{c\} \to \mathbf{N}$ mapping c to itself. But now, if $D := (C \ ; \ x_{77} \Leftarrow x_{1111})$, clearly D computes the same 1-variable function as C. And we have $\#C \in \mathcal{T}$, but $\#D \notin \mathcal{T}$ (simply because $\#D \neq c$), and so $\mathcal{T}$ is *not* a 1-index set.]

We shall complete our proof below by contradicting the undecidability of $\mathcal{T}$. (Actually the function displayed above will turn out to be the characteristic function of $\mathcal{T}$, or of its complement.)

Case 1. If, $\forall b \in \mathcal{S}$, domain(command$\#b) \neq \emptyset$, choose the fixed c_0 inside $\mathcal{S}$ [possible since the latter is non-empty]. Then

$$c \in \mathcal{T} \Longrightarrow [D_c \text{ computes the same one variable function as } C_0] \Longrightarrow \#D_c \in \mathcal{S} \ .$$

$$c \notin \mathcal{T} \Longrightarrow \ [\text{one variable function which } D_c \text{ computes has empty domain}] \ \Longrightarrow \#D_c \notin \mathcal{S} \ .$$

The left-hand "$\Longrightarrow$" each time is immediate from examining the command D_c . The other one uses 1-indexivity of $\mathcal{S}$.

Thus

$$c \in \mathcal{T} \Longleftrightarrow \#D_c \in \mathcal{S}$$

By $(*)$, this contradicts the undecidability of $\mathcal{T}$.

Case 2. If, $\exists b \in \mathcal{S}$ with domain(command$\#b) = \emptyset$, choose the fixed c_0 *not* in $\mathcal{S}$ [possible since the latter is a proper subset of $COM^{-1}(1)$]. Then

$$c \in \mathcal{T} \Longrightarrow [D_c \text{ computes the same one variable function as } C_0] \Longrightarrow \#D_c \notin \mathcal{S} \ .$$

$$c \notin \mathcal{T} \Longrightarrow \text{ [one variable function which } D_c \text{ computes has empty domain] } \Longrightarrow \#D_c \in \mathcal{S} \ .$$

The left-hand "$\Longrightarrow$" each time is immediate from examining the command D_c . The other one uses 1-indexivity of $\mathcal{S}$.

Thus

$$c \notin \mathcal{T} \Longleftrightarrow \#D_c \in \mathcal{S}$$

By $(*)$, this also contradicts the undecidability of $\mathcal{T}$.

IV-7: A Glimpse of Kleene's Creation : Recursion Theory.

We shall continue doing only the versions involving one variable functions of natural numbers. Several applications will be given before proving the two theorems below. See also **Theorem IV-9.1** for a really nice 'practical' application of the following.

Theorem IV-7.1. (Fixed Point Theorem.) *Suppose that $f : D \to \mathbf{N}$ is a recursive function such that $COM^{-1}(1) \subset D$ and $f(COM^{-1}(1)) \subset COM^{-1}(1)$* (so f is defined at least on all codes of commands and maps them to codes of commands). *Then f has a 'fixed point' in $COM^{-1}(1)$ in the following sense : $\exists c \in COM^{-1}(1)$ such that c and $f(c)$ code two commands which compute the* same *function of one variable.*

Application. *Among the* **BTEN** *commands which happen to compute total* constant *functions of one variable, there is at least one where the constant is actually the code number of the command itself!* (This is a 'self-actuating inattentive' command!)

To see this, take $f(b) =$ code of $[x_0 \leftarrow$ "b"$]$, for all $b \in \mathbf{N}$. Then f is certainly total and maps all of $\mathbf{N}$ to $COM^{-1}(1)$. It is seen to be recursive by simply writing down the code explicitly:

$$f(b) \;=\; CTR(0, CTR(0, 0, 1), SYN(b)) \; .$$

So we can choose C from **BTEN** such that the code of C, say c, is a fixed point. Since the command used in defining f clearly computes the total constant function with value b, it now follows that C computes exactly that function with $b = c$, as required.

Example. By this time, some readers who think like the author may be thinking : "Why can't I just define $f \;:\; COM^{-1}(1) \to COM^{-1}(1)$ by

$$f(\text{code of } C) \;:=\; \text{code of } (C \;;\; x_0 \leftarrow x_0 + 1) \quad .$$

This changes all the values of the function computed by C, by adding 1. Doesn't this contradict the theorem?" But no, in this case, the fixed points are all the codes of commands which compute the empty function.

Theorem IV-7.2. (Recursion Theorem.) *Suppose that $h : D \to \mathbf{N}$ is a partial recursive function of two variables. Then there is a* **BTEN** *command C such that the one variable function computed by C is $b \mapsto h(\#C, b)$.* (This is a drastic generalization of primitive recursion.)

Application. *Among the* **BTEN** *commands which happen to compute functions of one variable that have only one element in their domains, there is at least one where that number is actually the code number of the command itself!* (We can make it so that the function maps that code number to itself, and accuse the command of being 'narcissistically self-aware'! The cynic desiring to needle the artificial intellgensia might remark here that the theory of computationers have, with this one result, achieved far more, having simulated the infantile thought processes of 3/4 of the human population!)

To see this, take C to be the command obtained by applying the theorem to the obviously recursive

$$h \ : \ (a,b) \ \mapsto \ \begin{cases} b & \text{if } a = b \ ; \\ \text{undefined} & \text{if } a \neq b \ . \end{cases}$$

So, when the input is $(0, b, \underline{0})$ with $b \neq \#C$, the command C gives an infinite loop, as required.

[A **BTEN** command computing h above is

$$\mathbf{ite}(x_1 \approx x_2)(x_0 \leftarrow x_1)(\mathbf{whdo}(x_0 \approx x_0)(x_{11} \leftarrow x_7)) \ \ .]$$

Relation to the halting problem. Let

$$T \ := \ \{ \ (a,b) \ : \ a \text{ is the code of a command which halts with input } b \ \} \ .$$

We know from Turing's big result **IV-1.5** that

$$HAL_1 \ : \ (a,b) \ \mapsto \ \begin{cases} 1 & \text{if } (a,b) \in T \ ; \\ 0 & \text{if } (a,b) \notin T \ , \end{cases}$$

is *not* recursive; whereas

$$g \ : \ (a,b) \ \mapsto \ \begin{cases} 1 & \text{if } (a,b) \in T \ ; \\ \text{undefined} & \text{if } (a,b) \notin T \ , \end{cases}$$

is recursive. It does follow from **VI-5.2**. since the set T is $\mathcal{RE}$ but not $\mathcal{R}$, that the following function cannot be recursive :

$$h \ : \ (a,b) \ \mapsto \ \begin{cases} \text{undefined} & \text{if } (a,b) \in T \ ; \\ 0 & \text{if } (a,b) \notin T \ . \end{cases}$$

This latter fact can now alternatively be proved as an application of the Recursion Theorem. This is a somewhat different way to prove the undecidability of halting, but not necessarily recommended, since the original proof is very basic and simple, in particular, not really depending on the recursivity of Kleene's KLN relation.

To see this, assume for a contradiction that the above h is recursive, so we can apply **IV-7.2** to it. The resulting C computes the 1-variable function $b \mapsto h(\#C, b)$. Thus

$$\text{command } C \text{ does } \textit{not} \text{ halt on input } b \iff h(\#C, b) \text{ is undefined} \ .$$

But, by the definition of h,

$$h(\#C, b) \text{ is undefined } \iff (\#C, b) \in T \iff C \textit{ does} \text{ halt on input } b \ .$$

The two theorems are rather easily shown to imply each other, so let's do that (half-redundantly), before proving one of them.

Proof that IV-7.1 implies IV-7.2. Let H strongly compute h. Define $f : \mathbf{N} \to \mathbf{N}$ by

$$f(a) \ := \ \text{the code of } (x_2 \leftarrow x_1 \ ; \ x_1 \leftarrow \text{“}a\text{”} \ ; \ H) \ .$$

Then f is certainly total and maps all of $\mathbf{N}$ to $COM^{-1}(1)$. It is seen to be recursive by writing down the code explicitly, which is a bit messy! So we can choose C from **BTEN** which computes the same 1-variable function as $(x_2 \leftarrow x_1 \ ; \ x_1 \leftarrow \text{“}\#C\text{”} \ ; \ H)$. But that last command clearly computes $b \mapsto h(\#C, b)$, as required.

Proof that IV-7.2 implies IV-7.1. Given f, define h by $h(a, b) = U_1(f(a), b)$, where U_1 is the universal function of one variable. [Recall that $U_1(\#D, b)$ is the value at b of the 1-variable function defined by command D, as long as D halts with input $(0, b, \underline{0})$, and $U_1(d, b)$ is undefined for all other (d, b) .]

Choose a command C as in **IV-7.2** for this h. Let $c = \#C$. Then $f(c) = \#B$ for some command B, by the assumptions about f in the statement of **IV-7.1** . By **IV-7.2**, the 1-variable function defined by C is

$$b \ \mapsto \ h(\#C, b) = U_1(f(\#C), b) = U_1(f(c), b) = U_1(\#B, b) \ .$$

But $U_1(\#B, b)$ is what the 1-variable function defined by B maps b to. Thus c and $f(c)$ are codes of commands computing the same function, as required.

Proof of IV-7.1. Define $s : \mathbf{N} \to \mathbf{N}$ by letting $s(b)$ be the code of the following command, where N is sufficiently large, independently of b, and $C[U_1]$ is any universal command, that is, it strongly computes U_1 :

$$\begin{array}{l} (x_N \leftarrow x_1 \ ; \\ \ x_1 \leftarrow \text{“}b\text{”} \ ; \\ \ x_2 \leftarrow \text{“}b\text{”} \ ; \\ \ C[U_1] \ ; \\ \ x_1 \leftarrow x_0 \ ; \\ \ x_2 \leftarrow x_N \ ; \\ \ C[U_1]) \ . \end{array}$$

Once again, s is certainly total and maps all of $\mathbf{N}$ to $COM^{-1}(1)$. It is seen to be recursive by writing down the code explicitly, which is more than a bit messy!

The composite $f \circ s$ is then recursive, so let $C[f \circ s]$ be a command computing it, and let c_0 be the code of that command. The claim is that $s(c_0)$ is the ‘fixed point’.

Certainly both $s(c_0)$ and $f(s(c_0))$ are codes of commands. Now we combine the following three facts :

$$b \mapsto U_1(\#D, b) \text{ is the one variable function computed by } D \qquad (*)$$

[by the definition of U_1] ;

$$\forall b \quad U_1(c_0, b) \ = \ f(s(b)) \qquad (**)$$

[by $(*)$ and the definition of c_0] ;

$$\forall b \ \text{ the function computed by the command with code } s(b) \text{ is } d \mapsto U_1(U_1(b, b), d) \qquad (***)$$

[by directly examining the command above in the definition of s].

Thus the *1-variable function computed by the command with code* $f(s(c_0))$ is

$$\begin{aligned} b \quad &\mapsto \quad U_1(f(s(c_0), b) \qquad \text{by } (*) \\ &= U_1(U_1(c_0, c_0), b) \qquad \text{by } (**) \end{aligned}$$

which, by $(***)$, agrees with *1-variable function computed by the command with code* $s(c_0)$, as required.

IV-8: Universal Commands.

Despite the existence of universal functions of any number of variables (see **IV-1.3**), and the sense that the 2-variable universal function is 'totally universal' (see **IV-1.4**), we have the following easy non-existence result, for a command whose semantics resembles the universal functions.

There is no command V *such that, for all* $\underline{v} = (v_0, v_1, \cdots)$,

$$||V||(\underline{v}) \;=\; \begin{cases} \textit{err} & \text{if } v_0 \text{ is not a command \# ;} \\ ||\text{command}\#v_0||(v_1.v_2, \cdots) & \text{otherwise .} \end{cases}$$

If V existed, there would be some $i > 0$ such that $||V||(\underline{v})_i = v_i$ for all $\underline{v}$ such that $||V||(\underline{v}) \neq \textit{err}$: just take any $i > 0$ for which x_i doesn't appear in V (which we shall write as $x_i \notin V$). But letting v_0 be the number of a command which 'does nothing' (for example, $x_7 \leftarrow x_7$), note that $||V||$ would 'shift *every* slot one place to the left', contradicting the above.

Even if we restrict to $\underline{v}$ in which all but finitely many v_i are zero, essentially the same argument applies. Just choose a suitable $\underline{v}$ with non-zero v_i's for i well beyond the largest j for which $x_j \in V$.

Closely related, given an infinite sequence $B_1, B_2, \cdots$ of commands such that $v \mapsto \#B_v$ is computable, and given some $N > 0$, it is not necessarily the case that there is a command B such that $||B||(\underline{v}) = ||B_{v_N}||(\underline{v})$ for all $\underline{v}$. Just take B_v to be $x_v \leftarrow 1 + x_v$ (for all v). But now, choosing k larger than N, and putting k in the Nth slot,

$$||B||(\vec{0}, k, \underline{0})_k \;=\; ||B_k||(\vec{0}, k, \underline{0})_k \;=\; ||x_k \leftarrow 1 + x_k||(\vec{0}, k, \underline{0})_k \;=\; 1 \;\neq\; 0 \;=\; (\vec{0}, k, \underline{0})_k \;,$$

implying that $x_k \in B$. Since k can be arbitrarily large, the same contradiction as before occurs.

However, we do have such a B as long as all the B_v's together only 'use' a finite number of variables, and N is large. This theorem will be helpful in the next section.

Theorem IV-8.1. *Given* $N > 0$ *and an infinite sequence* $B_1, B_2, \cdots$ *from* **BTEN** *such that* $v \mapsto \#B_v$ *is a (total 1-variable) computable function, assume also that* $x_i \notin B_v$ *for all* $i \geq N$ *and for all* v. *Then there is a* B *in* **BTEN** *such that* $||B||(\underline{v}) = ||B_{v_N}||(\underline{v})$ *for all* $\underline{v}$.

One could, for example, make an effective list of *all* **BTEN** commands which use only the variables $x_0, x_1, \cdots, x_{N-1}$. The corresponding B would then be universal, in a sense, for such commands.

Two propositions will be used in the proof of the theorem.

Proposition IV-8.2 *If Dom* $\subset \mathbf{N}$ *and* $f : Dom \to \mathbf{N}$ *is a 1-variable* $\mathcal{BC}$*-function, then there is a command* D *in* **BTEN** *such that*

$$||D||(\underline{v}) \;=\; \begin{cases} err & \text{if } v_1 \notin Dom\ ; \\ (v_0, f(v_1), v_2, v_3, \cdots) & \text{if } v_1 \in Dom\ . \end{cases}$$

(So D must simply single out the second slot, i.e. slot # 1, compute f on it if possible, place the answer (if any) in slot #1, and restore all the other slots to their original values.)

Proof. Let $C[f]$ strongly compute f. Choose N such that $x_i \notin C[f]$ for $i > N$. Then let

$$D \;=\; \mathcal{B}_{0,2,3,\cdots,N}(\ C[f]\ ;\ x_1 \leftarrow x_0\ )\mathcal{E}\ .$$

Proposition IV-8.3 *For all* $N > 0$ *there is a command* C_N *in* **BTEN** *such that*

$$||C_N||(\underline{v}) = err \iff [\ COM(v_1) = 0 \text{ or } ||\text{command}\#v_1||(v_2, v_3, \cdots, v_{N+1}, \underline{0}) = err\ ],$$

and, when $||C_N||(\underline{v}) \neq err$,

$$||C_N||(\underline{v})_0 = ||\text{command}\#v_1||(v_2, v_3, \cdots, v_{N+1}, \underline{0})_0$$

(So C_N is a slight variation on commands which compute the various universal functions U_N of "$N+1$"-variables.)

Proof. It is easy to see that the following ϕ is a (total, 1-variable) computable function —just write out the number explicitly :

$$\phi\ :\ s\ \mapsto\ \#(\ x_0 \leftarrow x_1\ ; x_1 \leftarrow x_2\ ; \cdots ; x_{N-1} \leftarrow x_N\ ; x_N \leftarrow 0\ ; \text{command}\#s\ )\ .$$

In particular, $s \in \text{domain}(\phi) \iff COM(s)$.

Thus, the following ψ is a (total, "$N+1$"-variable) computable function, where U_N is a universal function of "$N+1$"-variables.

$$\psi\ :\ (v_1, v_2, \cdots, v_{N+1})\ \mapsto\ U_N(\phi(v_1), v_2, \cdots, v_{N+1})$$

Let C_N be any **BTEN**-command which strongly computes ψ. Then,

$$||C_N||(\underline{v}) \neq err \iff [\ v_1 \in \text{domain}(\phi) \text{ and } (\phi(v_1), v_2, \cdots, v_{N+1}) \in \text{domain}(U_N)\ ]$$

$$\iff [\ COM(v_1) \neq 0 \text{ and } ||\text{command}\#\phi(v_1)||(0, v_2, v_3, \cdots, v_{N+1}, \underline{0}) \neq err\ ]$$

$$\iff [\ COM(v_1) \neq 0 \text{ and}$$

$$||(\ x_0 \leftarrow x_1\ ; x_1 \leftarrow x_2\ ; \cdots ; x_{N-1} \leftarrow x_N\ ; x_N \leftarrow 0\ ; \text{command}\#v_1\ )||(0, v_2, v_3, \cdots, v_{N+1}, \underline{0}) \neq err\ ]$$

$$\iff [\ COM(v_1) \neq 0 \text{ and } ||\text{command}\#v_1||(v_2, v_3, \cdots, v_{N+1}, \underline{0}) \neq err\ ],$$

as required. Furthermore, when $||C_N||(\underline{v}) \neq err$, we have

$$||C_N||(\underline{v})_0 \;=\; \psi(v_1,\cdots,v_{N+1}) \;=\; U_N(\phi(v_1),v_2,\cdots,v_{N+1})$$

$$=\; ||\text{command}\#\phi(v_1)||(0,v_2,v_3,\cdots,v_{N+1},\underline{0})_0$$

$$=\; ||(\; x_0 \leftarrow x_1 \,;x_1 \leftarrow x_2 \,;\cdots;x_{N-1} \leftarrow x_N \,;x_N \leftarrow 0 \,;\text{command}\#v_1\;)||(0,v_2,v_3,\cdots,v_{N+1},\underline{0})_0$$

$$=\; ||\text{command}\#v_1\;)||(v_2,v_3,\cdots,v_{N+1},\underline{0})_0 \;,\quad \text{as required .}$$

Proof of Theorem IV-8.1. Fix i with $0 \leq i < N$, and in **IV-8.2** take

$$f \;=\; f_i \;:\; v_i \;\mapsto\; \#(\;\text{command}\#v_1 \;;\; x_0 \leftarrow x_i \;)\;.$$

So we get a command D_i with

$$||D_i||(\underline{v}) \;=\; \begin{cases} err & \text{if } COM(v_1)=0\;; \\ (v_0,f_i(v_1),v_2,v_3,\cdots) & \text{otherwise .} \end{cases}$$

Now define $E_{N,i} := (D_i \;;\; C_N)$, for C_N from **IV-8.3**. Then

$$||E_{N,i}||(\underline{v}) \;=\; ||C_N||(||D_i||(\underline{v})) \;=\; ||C_N||(\text{the RHS in the display just above})\;.$$

Thus

$$||E_{N,i}||(\underline{v}) = err \Longleftrightarrow [\; COM(v_1)=0 \;\text{ or }\; ||\text{command}\#v_1||(v_2,v_3,\cdots,v_{N+1},\underline{0}) = err\;] \qquad (*)$$

If $||E_{N,i}||(\underline{v})) \neq err$, then

$$||E_{N,i}||(\underline{v})_0 \;=\; ||C_N||(v_0,f_i(v_1),v_2,v_3,\cdots)_0$$

$$=\; ||\text{command}\#f_i(v_1)||(v_2,v_3,\cdots,v_{N+1},\underline{0})_0$$

$$=\; ||(\;\text{command}\#v_1 \;;\; x_0 \leftarrow x_i \;)||(v_2,v_3,\cdots,v_{N+1},\underline{0})_0$$

$$=\; ||\text{command}\#v_1||(v_2,v_3,\cdots,v_{N+1},\underline{0})_i \qquad (**)$$

Now let A be a command strongly computing $v \mapsto \#B_v$. Choose $M > N+1$ such that $x_j \notin A$ for $j \geq M$, and also $x_j \notin D_0 \cup D_1 \cup \cdots \cup D_{N-1} \cup C_N$ for $j > M+N$. Define B as follows:

$$\mathcal{B}_{N,N+1,\cdots,N+M}(x_2 \leftarrow x_0 \;;\; x_3 \leftarrow x_1 \;;\; \cdots \;;\; x_{N+1} \leftarrow x_{N-1} \;;\; x_1 \leftarrow x_N \;;\; \mathcal{B}_{2,3,\cdots,N+1}A\mathcal{E} \;;\; x_1 \leftarrow x_0 \;;$$

$$\mathcal{B}_{1,2,\cdots,M+N}E_{N,0}\mathcal{E} \;;\; x_M \leftarrow x_0 \;;$$

$$\mathcal{B}_{1,2,\cdots,M+N}E_{N,1}\mathcal{E} \;;\; x_{M+1} \leftarrow x_0 \;;$$

$$\vdots$$

$$\mathcal{B}_{1,2,\cdots,M+N}E_{N,N-1}\mathcal{E} \;;\; x_{M+N-1} \leftarrow x_0 \;;$$

$$x_0 \leftarrow x_M \;;\; x_1 \leftarrow x_{M+1} \;;\; \cdots \;;\; x_{N-1} \leftarrow x_{M+N-1})\mathcal{E}$$

To prove the theorem, i.e. that $||B||(\underline{v}) = ||B_{v_N}||(\underline{v})$, consider three cases:

For those $\underline{v}$ with $||B_{v_N}||(\underline{v}) \neq err$ and for $i \geq N$,
$||B_{v_N}||(\underline{v})_i = v_i$ since $x_i \notin B_{v_N}$ for these i ; whereas $||B||(\underline{v})_i = v_i$ since B is a command starting with the symbol $\mathcal{B}_{N,N+1,\cdots,N+M}$ and since $x_i \notin B$ for $i > M+N$.

For those $\underline{v}$ with $||B_{v_N}||(\underline{v}) \neq err$ and for $i < N$, one just follows the command B and uses $(**)$ above to see that $||B||(\underline{v})_i = ||B_{v_N}||(\underline{v})_i$. The first displayed line of B 'takes' $\underline{v}$ to $(\#B_{v_N}, \#B_{v_N}, v_0, v_1, \cdots, v_{N-1}, - - -)$, where the $- - -$ indicates that we are not interested in those slots. Consider all the middle lines (before the last line of B), of the form

$$\mathcal{B}_{1,2,\cdots,M+N} E_{N,i} \mathcal{E} \ ; \ x_{M+i} \Leftarrow x_0 \ ;$$

for $0 \leq i \leq N-1$. By $(**)$, they take the latter slot entries to

$$(--, ||B_{v_N}||(v_0, \cdots, v_{N-1}, \underline{0})_0, ||B_{v_N}||(v_0, \cdots, v_{N-1}, \underline{0})_1, \cdots, ||B_{v_N}||(v_0, \cdots, v_{N-1}, \underline{0})_{N-1}, --)$$

where the indicated slots are the Mth to the $(M+N-1)$th. Finally the last line in B shifts all these to the left as far as they will go; that is, $||B_{v_N}||(v_0, \cdots, v_{N-1}, \underline{0})_i$ is in the ith slot. But the latter number is in fact $||B_{v_N}||(\underline{v})_i$, as required, since the command B_{v_N} contains no variables x_j for $j \geq N$.

For those $\underline{v}$ with $||B_{v_N}||(\underline{v}) = err$, following the command B, we get err as soon as the subcommand $E_{N,0}$ on the second line is reached, using the previous display $(*)$, since the latter subcommand will be acting with input $(-, \#B_{v_N}, v_0, v_1, \cdots, v_{N-1}, - - -)$. Note here that $x_i \in B_{v_N}$ at most for $i = 0, 1, \cdots, N-1$, so the $-$'s are irrelevant.

IV-9: Fixed Point Equations for Commands— a.k.a. "Recursive Programs".

Those persons, who know more about programming than I, will have seen plenty of instances of programs written somewhat in the following style:
"I am a program whose name is NAME, and here I am:
(——;NAME;—;NAME;——;NAME;—) "
In other words, the name of the 'whole' program occurs one or more times *within* the program itself, as though it were a 'macro' or 'subroutine' within itself. This is a somewhat novel kind of self-reference compared to what we see in other parts of logic and mathematics. It appears to give a new way of 'looping' within programs, different from what we've seen so far in this paper, where non-termination of a computation always has been caused by a **whdo**-command. This self-reference is certainly not part of the rules for producing syntactically valid commands in any of our languages **ATEN**, etc. And it is interesting to give a rigorous version of the semantics of such programs, and to show that there is always a genuine **ATEN**-command which has the same semantics. The latter must be true, or else we would have a contradiction to Church's thesis, since we know that every recursive function can be computed by some **ATEN**-command.

Actually, this can be done more generally, by having several programs with specified names, with each program containing the names, once or several times, of some or all

of these programs. One generalizes the theory below, using several 'program variables', $Z_1, Z_2, \cdots$; and a system of 'program equations', $Z_1 \equiv C_1 \ // \ Z_2 \equiv C_2 \ // \ \cdots$. We shall stick to the 1-variable case.

First, here is slight extension of the language **ATEN** to a language **ZTEN**. We have decided, so to speak, to always use Z as the "NAME" in the discussion above. Syntactically, we just add Z on its own as a new atomic 'command', to the usual inductive rules. So **ZTEN** is the smallest set of strings of symbols containing :

$$Z \ // \ x_i \Leftarrow t \ // \ (C; D) \text{ and } \mathbf{whdo}(F)(C) \ , \text{ whenever } C \text{ and } D \text{ are in } \mathbf{ZTEN} \ ,$$

with t and F from the language of $1^{\underline{\text{st}}}$order number theory, as before.

On its own, this is not enough. We wish to give semantic meaning to expressions of the form $Z \equiv C$, for any C in **ZTEN**, in a way which reflects our intuition of what is going on. The "$\equiv$"-symbol is supposed to denote requiring that this is 'an equation up to semantics'. The string C is not going to be the same string as the single symbol Z. But a 'solution' to the equation should be some non-mysterious object, such as an **ATEN**-command D, which has the same computational effect as the longer command you'd get by substituting D for every occurrence of Z in C.

Roughly speaking, what is going on is that you start computing with C and with an input from $\mathbf{N}^\infty$ as usual. As soon as you reach any occurrence of Z within C, you go back to the beginning of C and continue making computational steps. The point is that each time you return to the beginning of C, the state of the bins, i.e. the new input, may very well be different than any that has occurred in previous returns to the beginning. And the 'hope' is that eventually, you will get to the end of C, 'getting beyond' occurrences of Z, and that the computation will then terminate. (Of course, it might do an infinite loop in the *usual* way as well.) From this description, it certainly appears that we are dealing with a perfectly determinate effective mechanical process, as described intuitively at great length early in this paper.

An example from programming practice might look a bit like the following:

My name is FACT and here I am:

if $x_1 \approx 0$
then $x_0 \Leftarrow 1$
else($x_2 \Leftarrow x_1$;
$x_1 \Leftarrow pred(x_1)$;
FACT ;
$x_0 \Leftarrow x_2 \times x_0$)

Here we are using *pred* for the predecessor function which fixes 0 and subtracts 1 from everything else, so that we're using several shortcuts, including **BTEN** rather than **ATEN**. The 'user' must also be made to realize that x_1 is the input and x_0 is the output, and 'therefore', each time you 'looped back' into FACT, the name of the variable x_2 must be changed. If you input $x_1 = 4$ and follow it through with the proviso above, the program will terminate, after looping four times, with output 24. And that 24 is obtained by successively subtracting 1, starting with 4, and multiplying all the results together, i.e. producing 4! . A bit of thought leads one to conclude that this is a neat little program for calculating the factorial function.

Now the style we'll use for this is the following, where Z is used systematically, and the $\mathcal{B}\cdots\mathcal{E}$-construction from **BTEN** is used to avoid the need to specify input-output, and the corresponding need to 'upgrade' the names of variables which occur peripherally in the calculation. Here is the previous, done now in this style.

$Z \equiv$ **if** $x_1 \approx 0$
then $x_0 \Leftarrow 1$
else($x_2 \Leftarrow x_1$;
$x_1 \Leftarrow pred(x_1)$;
$\mathcal{B}_2 Z \mathcal{E}$;
$x_0 \Leftarrow x_2 \times x_0$)

Call the **ZTEN**-string on the RHS of $\equiv$ by the name C. If we substitute C itself for the Z in C, then do it again a few times, we end up with the following much larger string from **ZTEN**. (I'm referring to the 'command' on the left, not the computational history recorded on the right.) This is what is further down called $A_5(C)$. It could be defined inductively either as '$A_4(C)$ with C substituted for Z', or as 'C with $A_4(C)$ substituted for Z'. Below, these are denoted as $A_4(C)^{[Z\rightarrow\ C]}$ and $C^{[Z\rightarrow\ A_4(C)]}$, respectively. In fact, it will also be the same 'Z-string' as $A_2(C)^{[Z\rightarrow\ A_3(C)]}$, etc.

if $x_1 \approx 0$ ———(0 , 4 , 0 , ·····) Note that $v_1 = 4$—variables start with x_0!
then $x_0 \Leftarrow 1$
else($x_2 \Leftarrow x_1$; ———(0 , 4 , 4 , ·····)
$x_1 \Leftarrow pred(x_1)$; ———(0 , 3 , 4 , ·····)
$\underline{\underline{\mathcal{B}_2}}$ **if** $x_1 \approx 0$
then $x_0 \Leftarrow 1$
else($x_2 \Leftarrow x_1$; ———(0 , 3 , 3 , ·····)
$x_1 \Leftarrow pred(x_1)$; ———(0 , 2 , 3 , ·····)
$\underline{\underline{\mathcal{B}_2}}$ **if** $x_1 \approx 0$
then $x_0 \Leftarrow 1$
else($x_2 \Leftarrow x_1$; ———(0 , 2 , 2 , ·····)
$x_1 \Leftarrow pred(x_1)$; ———(0 , 1 , 2 , ·····)
$\underline{\mathcal{B}_2}$ **if** $x_1 \approx 0$
then $x_0 \Leftarrow 1$
else($x_2 \Leftarrow x_1$; ———(0 , 1 , 1 , ·····)
$x_1 \Leftarrow pred(x_1)$; ———(0 , 0 , 1 , ·····)
$\mathcal{B}_2$ **if** $x_1 \approx 0$
then $x_0 \Leftarrow 1$ ———(1 , 0 , 1 , ·····)
else($x_2 \Leftarrow x_1$;
$x_1 \Leftarrow pred(x_1)$;
$\mathcal{B}_2 Z \mathcal{E}$;
$x_0 \Leftarrow x_2 \times x_0$) $\mathcal{E}$; ———(1 , 0 , 1 , ·····)
$x_0 \Leftarrow x_2 \times x_0$) $\underline{\mathcal{E}}$; ———(1·1 , 0 , 2 , ·····)
$x_0 \Leftarrow x_2 \times x_0$) $\underline{\underline{\mathcal{E}}}$; ———(1·1·2 , 0 , 3 , ·····)
$x_0 \Leftarrow x_2 \times x_0$) $\underline{\underline{\underline{\mathcal{E}}}}$; ———(1·1·2·3 , 0 , 4 , ·····)
$x_0 \Leftarrow x_2 \times x_0$) ———(1·1·2·3·4 , − , − , ·····) = (24 , − , − , ·····)

On the right is the computational history which would result from starting with input $(0, 4, 0, 0, \cdots)$, if we just went ahead and computed, not noticing the "Z" buried in the middle of the string, and assuming the string was actually some **BTEN**-command. I have single-, double-, and triple-underlined the corresponding $\mathcal{B}$'s and $\mathcal{E}$'s which actually affect this calculation, restoring the 2, 3, and 4 into slot # 2, respectively.

The main point to notice is that the Z never came into play in this computation—in fact, we could have just used $A_4(C)$ to get the same result; that is, we did one more substitution (of C for Z) than strictly necessary. And the string above would have produced a resulting output of 5! in slot # 0 if we'd started with input $(0, 5, 0, 0, \cdots)$. But with input $(0, 6, 0, 0, \cdots)$, there would be a conundrum with the above string, when we reached the symbol Z. (Actually, using the semantics of **ZTEN** as defined below, we'd just get err.) For that input, we'd need $A_n(C)$ for any $n \geq 6$, and the computation would always produce a resulting output of 6! .

At this point, there should be enough motivation for two definitions.

Definition of semantics for ZTEN. Use exactly the same formulas as in the semantics of **BTEN**, but add the stipulation that $||Z||(\underline{v}) = err$ for all $\underline{v}$.

Definition of substitution. If C is a string from **ZTEN**, and D is also, define $C^{[Z\to\ D]}$ by induction on C as follows:

$$Z^{[Z\to\ D]} \ := \ D$$

$$x_i \Leftarrow t^{[Z\to\ D]} \ := \ x_i \Leftarrow t$$

$$(C_1; C_2)^{[Z\to\ D]} \ := \ (\ C_1^{[Z\to\ D]} \ ; \ C_2^{[Z\to\ D]} \ )$$

$$\mathbf{whdo}(F)(C)^{[Z\to\ D]} \ := \ \mathbf{whdo}(F)(C^{[Z\to\ D]}) \ .$$

The following are then very easy to prove by induction on C:

$C^{[Z\to\ D]}$ is a string in **ZTEN** .

If D is in **BTEN**, *then $C^{[Z\to\ D]}$ is in fact a command in* **BTEN** .

Now, further down, we shall pursue carefully this approach with these recursive programs, and show quite explicitly how to produce an actual **BTEN**-command which reproduces the semantics (yet to be defined!) of a 'whatever' produced by an 'equation', $Z \equiv C$, for C from **ZTEN**. In fact, $A_\infty(C)$ would turn out to be a good name for whatever it is that $Z \equiv C$ defines. We could make a perfectly good *doubly infinite* sequence of **BTEN**-symbols with the name $A_\infty(C)$. But that of course is not in **BTEN**. And it will still remain to produce a **BTEN**-command with the same semantics.

But if by "semantics" we mean that we're only interested in the 1-variable function computed by these things, it is easy to explain what is wanted, without all these iterated substitutions. And a very elegant application of Kleene's Fixed Point Theorem proves the existence of a **BTEN**-command which computes the same 1-variable function. Actually,

Kleene's Theorem works for any number of variables, so what we do just below can be made as general as needed.

Theorem IV-9.1. *For any C from* **ZTEN**, *there is a command D from* **BTEN** *such that the 1-variable functions defined by D, and by $C^{[Z\to\ D]}$, are the same .* (Since $Z^{[Z\to\ D]} = D$, this says that D is a solution to $Z \equiv C$, if by "a solution", we mean a command such that the two sides of the equation produce the same 1-variable function, when that command is subsituted for Z.)

Proof. Consider the function $f_C : COM^{-1}(1) \to \mathbf{N}$ defined by $f_C(\#D) := \#C^{[Z\to\ D]}$. By Kleene's theorem, **IV-7.1**, we need only verify that f_C is recursive, in order to then conclude as required. This verification is straightforward by induction on C. For f_C is a constant or identity function when C is atomic ; $f_{(C_1;C_2)} = CTR(2, f_{C_1}, f_{C_2})$; and $f_{\mathbf{whdo}(F)(C)} = CTR(3, \#F, f_C)$.

So that was certainly easy! But let's go back to pursuing the more direct, pedestrian approach.

Prop. IV-9.2. *We have $C^{[Z\to\ D^{[Z\to\ E]}]} = (C^{[Z\to\ D]})^{[Z\to\ E]}$ for all C, D, E in* **ZTEN** .

Proof. This is straightforward by induction on C.

Now here is the formal definition which was illustrated lavishly earlier with $n = 5$ and a particular C.

Definition. Given C in **ZTEN**, define $A_n(C)$ in **ZTEN** for $n \geq 0$ by induction on n as follows:

$$A_0(C) \ := \ Z \quad ; \quad A_{k+1}(C) \ := \ A_k(C)^{[Z\to\ C]} \ .$$

Note that $A_1(C) \ = \ Z^{[Z\to\ C]} \ = \ C$.

Cor. IV-9.3. *We have $A_i(C)^{[Z\to\ A_j(C)]} = A_{i+j}(C)$ for all $i, j \geq 0$.*

(In particular, $A_{k+1}(C)$ is also $C^{[Z\to\ A_k(C)]}$.)

Proof. Proceed by induction on j, using **IV-9.2** .

Prop. IV-9.4. *For any C, D in* **ZTEN**,

$$||D||(\underline{v}) \neq err \implies ||D^{[Z\to\ C]}||(\underline{v}) = ||D||(\underline{v}) \ .$$

(The intuition here of course is that, by the hypothesis, the computation using D with input $\underline{v}$ will never get tangled up with any Z's that occur in D. And so substituting something for Z in D and using that instead will produce the same computation as the one using D.)

Proof. Proceed by induction on D, simultaneously for all $\underline{v}$.

For $D = Z$: There are no such $\underline{v}$.

For $D = x_i \leftarrow t$: Here $D^{[Z\to\ C]} \ = \ D$.

For $D = (D_1; D_2)$: Then $||D_1||(\underline{v}) \neq err$ and $||D_2||(||D_1||(\underline{v})) \neq err$. So

$$||D^{[Z\to\ C]}||(\underline{v}) \;=\; ||(D_1^{[Z\to\ C]}\;;\; D_2^{[Z\to\ C]})||(\underline{v})) \;=\; ||D_2^{[Z\to\ C]}||(||D_1^{[Z\to\ C]}||(\underline{v}))$$

$$||D_2^{[Z\to\ C]}||(||D_1||(\underline{v})) \;=\; ||D_2||(||D_1||(\underline{v})) \;=\; ||D||(\underline{v})\ .$$

Of course, we just used the inductive hypotheses for both D_i's.

For $D = \mathbf{whdo}(F)(E)$: Here the result is assumed for E in place of D. Since $||D||(\underline{v}) \neq err$, by the definition of the semantics of **whdo**-commands, there is some $k \geq 0$ such that $||E||^j(\underline{v}) \neq err$ for $0 \leq j \leq k$, and the formula F is true at $||E||^j(\underline{v})$ for $0 \leq j < k$, but false at $||E||^k(\underline{v})$. Furthermore, $||D||(\underline{v}) = ||E||^k(\underline{v})$.

In the extreme case when $k = 0$, the formula F is false at $\underline{v}$, and we get $||D||(\underline{v}) = \underline{v}$. But

$$||D^{[Z\to\ C]}||(\underline{v}) = ||\mathbf{whdo}(F)(E^{[Z\to\ C]})||(\underline{v}) = \underline{v}$$

also, again since F is false at $\underline{v}$.

When $k > 0$, first, by induction on j, we prove that

$$||E^{[Z\to\ C]}||^j(\underline{v}) \;=\; ||E||^j(\underline{v}) \quad \text{for} \;\; 1 \leq j \leq k\ ,$$

(and so the 'truth and falsity of F' works the same for $E^{[Z\to\ C]}$ as for E). Here is the inductive step:

$$||E^{[Z\to\ C]}||^{j+1}(\underline{v}) \;=\; ||E^{[Z\to\ C]}||(||E^{[Z\to\ C]}||^j(\underline{v}))$$

(now using the inductive hypotheses concerning j, then concerning E)

$$=\; ||E^{[Z\to\ C]}||(||E||^j(\underline{v})) \;=\; ||E||(||E||^j(\underline{v})) \;=\; ||E||^{j+1}(\underline{v})\ .$$

So we get

$$||D^{[Z\to\ C]}||(\underline{v}) \;=\; ||\mathbf{whdo}(F)(E^{[Z\to\ C]})||(\underline{v}) \;=\; ||E^{[Z\to\ C]}||^k(\underline{v}) \;=\; ||E||^k(\underline{v}) \;=\; ||D||(\underline{v})\ ,$$

where the second equality uses the observation above about the 'truth and falsity of F'.

Theorem IV-9.5. *If $||A_n(C)||(\underline{v}) \neq err$, then we have $||A_k(C)||(\underline{v}) = ||A_n(C)||(\underline{v})$ for all $k \geq n$.*

Proof. It clearly suffices to establish this for $k = n + 1$, and the latter is immediate from **IV-9.4** .

Definition. For any C in **ZTEN**, let us simply regard "$Z \equiv C$" and "$A_\infty(C)$" as suggestive strings of symbols—they are not in any of the inductively defined sets **?TEN**. However, define their semantics (making use of **IV-9.5** to see the correctness of the definition) by

$$||Z \equiv C||(\underline{v}) \;:=\; ||A_\infty(C)||(\underline{v}) \;:=\; \begin{cases} err & \text{if } ||A_n(C)||(\underline{v}) = err \;\; \forall n\ ; \\ ||A_n(C)||(\underline{v}) & \text{if } ||A_n(C)||(\underline{v}) \neq err\ . \end{cases}$$

This clearly agrees with the earlier intuitively described process of successively substituting C 'into itself' for Z. And then trying to compute with a given input until (and if) you get to a stage where the computation never gets involved with any occurrence of Z in the **ZTEN**-string, and so is carried out in accordance with the operational semantics of **BTEN**.

So we've made the semantics rigorous. Finally, let's introduce one more clever idea from the CSer's, to give very direct construction of an actual **BTEN**-command whose semantics agrees exactly—in all detail, not just for the 1-variable function defined—with the semantics of $||Z \equiv C||$ as defined just above.

The idea is to replace Z everywhere in the $A_n(C)$ by a 'flagging variable' being set to 0. This is like a 'warning flag' which, when reached, warns that you must increase n to have any hope for the computation to 'converge'. We'll call the resulting strings $B_n(C)$. They have the advantage of being in **BTEN**, not just **ZTEN**, and so won't give infinite loops caused by running into Z.

Then the idea for the desired command in **BTEN** is simply to set up a command whose effect is to successively go through $B_n(C)$ for larger and larger n. Each time you hit the flagged variable set to zero, you start over with the next value of n. If and when you finally get to an n for which the flagged variable remains being 1, you just complete the computation using that particular $B_n(C)$. In this, the subscript n is itself behaving like a variable. To make this work, we use the theorem from the previous section.

Definition. Fix C in **ZTEN**. Choose N such that $i \geq N-1 \implies x_i \notin C$. Define $B_n(C)$ in **ZTEN** for $n \geq 0$ by induction on n as follows:

$$B_0(C) \;:=\; x_{N-1} \leftarrow 0 \;\;;\;\; B_{k+1}(C) \;:=\; C^{[Z \to\, B_k(C)]} \;.$$

It is clear that each $B_n(C)$ is in **BTEN**, and that $i \geq N \implies x_i \notin B_k(C)$. Furthermore, $B_n(C) = A_n(C)^{[Z \to\, x_{N-1} \leftarrow 0]}$, as easily follows using **IV-9.2** .

Lemma IV-9.6. *The function $v \mapsto \#B_v(C)$ is total recursive.*

Proof. First extend the Gödel numbering of **BTEN** to **ZTEN** by requiring

$$\#Z \;:=\; <4,0,0> \;,$$

keeping the same inductive formulas for non-atomic strings, or 'ZCommands', in **ZTEN**. The 'usual' argument (i.e. course-of-values-recursion/CLV) shows that the following is a primitive recursive relation, using obvious extensions of old notation:

$$H \;:\; (a,b,c) \;\mapsto\; \begin{cases} 1 & \text{if } ZCOM(a)\;,\; ZCOM(b) \text{ and } c = \#(\text{command}_Z \#a^{[Z \to\, \text{command}_Z \#b]}) \;; \\ 0 & \text{otherwise} \quad . \end{cases}$$

Thus, minimizing over c,

$$h \;:\; (a,b) \;\mapsto\; \#(\text{command}_Z \#a^{[Z \to\, \text{command}_Z \#b]}) \;=\; \min\{\; c \;:\; H(a,b,c) = 1 \;\}$$

is a partial recursive function. Now we have

$$\#B_{v+1}(C) \;=\; h(\#C, \#B_v(C)) \;.$$

And so the function $v \mapsto \#B_v(C)$ is obtained by primitive recursion from a recursive function.

Lemma IV-9.7. *There is a command $B(C)$ in* **BTEN** *such that, for all $\underline{v}$,*

$$||B(C)||(\underline{v}) = ||B_{v_N}(C)||(\underline{v}) \ .$$

Proof. This is immediate from **IV-8.1**, using **IV-9.6** .

Theorem IV-9.8. *For each C in* **BTEN** *there is a command D in* **BTEN** *such that, for all $\underline{v}$,*

$$||D||(\underline{v}) = ||Z \equiv C||(\underline{v}) = ||A_\infty(C)||(\underline{v}) \ .$$

This should be compared to **Theorem IV-9.1**, which it implies; and the proofs compared as well.

Proof. Choose $B(C)$ as in **IV-9.7**, and an integer $M \geq 1$ so that $x_i \notin B(C)$ for $i > N + M$. Now let D be the following command:

$$\begin{aligned} &\mathcal{B}_{N-1,N,N+1,\cdots,N+M}(x_N \leftarrow 0 \ ; \ x_{N-1} \leftarrow 0 \ ; \\ &\qquad\qquad \textbf{while } x_{N-1} \approx 0 \\ &\qquad\qquad \textbf{do } (\ x_N \leftarrow 1 + x_N \ ; \ x_{N-1} \leftarrow 1 \ ; \ \mathcal{B}_{0,1,\cdots,N-2}B(C)\mathcal{E}\ ) \ ; \\ &\qquad B(C))\mathcal{E} \end{aligned}$$

This is the command described further up, with x_{N-1} as the flag variable, and x_N playing the role of the "subscript". Now let's prove the required semantic equality in a more formal manner.

Observation. Letting $\underline{v}$ be the input into D, observe that, at every step in the computation before the final "$B(C)$" is executed (if ever), the state $\underline{w}$ satisfies $w_i = v_i$ for all $i < N - 1$. This is because both of the *explicit* x_i in D have $i \geq N - 1$, and because of the $\mathcal{B}_{0,1,\cdots,N-2}$. (Of course, we take $\mathcal{B}_{0,1,\cdots,N-2}B(C)\mathcal{E}$ s a unit, and do not stop to check the state here during the middle of $B(C)$'s execution, nor after doing $B(C)$ and before restoring the original values into $v_0, \cdots, v_{N-2}$!)

In the case of those $\underline{v}$ with $||A_\infty(C)||(\underline{v}) = err$:

By the definition of $||A_\infty(C)||$, we have $||A_k(C)||(\underline{v})) = err$ for all k. Thus, for all k,

$$||B_k(C)||(\underline{v}) \neq err \implies ||B_k(C)||(\underline{v})_{N-1} = 0 \ .$$

So, for all $\underline{w}$ with $w_i = v_i$ for all $i < N - 1$, we have

$$||B_k(C)||(\underline{w}) \neq err \implies ||B_k(C)||(\underline{w})_{N-1} = 0 \ .$$

Thus, for the same $\underline{w}$, we have

$$||B(C)||(\underline{w}) \neq err \implies ||B(C)||(\underline{w})_{N-1} = 0 \ .$$

And so, by the observation above, the **whdo**-loop in D is never escaped, and we find $||D||(\underline{v}) = err$, as required.

In the case of those $\underline{v}$ with $||A_\infty(C)||(\underline{v}) \neq err$:

By the definition of $||A_\infty(C)||$, there is a unique $m \geq 1$ with

$$||A_{m-1}(C)||(\underline{v}) \;=\; err \;\neq\; ||A_m(C)||(\underline{v}) \;=\; ||A_\infty(C)||(\underline{v}) \;.$$

Then, for all $\underline{w}$ with $w_i = v_i$ for all $i < N-1$, we have

$$||B_k(C)||(\underline{w}) \;=\; \begin{cases} err & \text{if } k < m \;; \\ ||A_m(C)||(\underline{w}) & \text{if } k \geq m \;. \end{cases}$$

In particular, note that, for such $\underline{w}$,

$$||B_m(C)||(\underline{w}) \;=\; ||A_\infty(C)||(\underline{w}) \qquad\qquad (*)$$

Thus the **whdo**-loop in D loops until $v_N = m$ is reached. Then we get the following.

For all $i < N-1$:

$||D||(\underline{v})_i$

$= ||B(C)||(\underline{w})_i$ for $\underline{w}$ with $w_j = v_j \; \forall j < N-1; w_{N-1} = 1; w_N = m; w_j = v_j \; \forall j > N+M$;

$= ||B_{w_N}(C)||(\underline{w})_i$ by the definition of $B(C)$ in **IV-9.7** ;

$= ||B_m(C)||(\underline{w})_i$ since $w_N = m$;

$= ||B_m(C)||(\underline{v})_i$ since $x_j \notin B_m(C) \; \forall j \geq N-1$ and $w_j = v_j \; \forall j < N-1$;

$= ||A_\infty(C)||(\underline{v})_i$ by $(*)$.

And for $N-1 \leq i \leq N+M$:

$||D||(\underline{v})_i \;=\; v_i$ since D starts with the symbol $\mathcal{B}_{N-1,N,\cdots,N+M}$;

$= ||B_m(C)||(\underline{v})_i$ since $x_j \notin B_m(C) \; \forall j \geq N-1$

and execution of $B_m(C)$ on $\underline{v}$ never reaches any of the $(x_{N-1} \Leftarrow 0)'$s ;

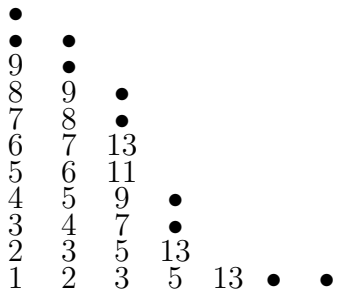
$= ||A_\infty(C)||(\underline{v})_i$ by $(*)$.

And for $i > N+M$:

Again, both clearly give v_i .

IV-10: Ackermann's Function.

The easiest way to remember this total function of two variables is geometrically. The array of its values is started below.



Think of filling it in column-by-column. The leftmost column (the 0th) is evident. For an entry in the 0th row, use the entry diagonally above and to the left. To find an entry in a box away from the edges, given all entries below it and in columns to its left, go down one box to see an entry, say, N. Then go to the Nth entry in the column immediately to the left, and that's the answer. (It's really the $(N+1)$st entry, since the rows are numbered from the 0th.) So of course, we only need finitely many entries in the column to the left (but often quite a few!), to get the entry we're after.

As a self-test, check that the first few entries along the main diagonal are

$$1 \ , \ 3 \ , \ 7 \ , \ 61 \ , \ 2^{2^{2^{2^{16}}}} - 3 \ .$$

This gives an alternative answer to the I.Q. question : 1 , 3 , 7 , $\underline{?}$, $\underline{?}$! It also perhaps hints at how fast Ackermann's function grows.

So here is the 'nested recursion' which summarizes the geometrical description above :

$$A(0,n) := n{+}1 \ ; \ A(m{+}1,0) := A(m,1) \ ; \ A(m{+}1,n{+}1) := A(m,A(m{+}1,n)) \ .$$

Ackermann's function played a major role in the history of computability. It alerted Hilbert et al to the situation that there would probably be recursions which 'went beyond' any fixed recursion scheme. (See however McCarthy's scheme in the next section.) It even led to some pessimism for awhile about the possibility of ever finding a satisfactory definition of *computability*. For us, it is (as per the theorem below) a concrete example of something we already know exists by an easier but less direct argument (see

the last page of **IV-4**) , as well as providing a nice excuse for illustrating the power of recursive programming.

Theorem IV-4.10.1 *Ackermann's function A is* (i) *total recursive, but* (ii) *not primitive recursive.*

Proof of (i). If we permit Church's thesis, this is clear from the description above, which gives a mechanical method for producing any of its values. Directly constructing A as a recursive function seems to be very tedious. A specific construction of a **BTEN**-command for computing it was found in a tour-de-force by Ben Willson in [**W**].

However, the computation of A provides a perfect candidate for using **ZTEN** from the previous section. First, here is a re-write of the nested recursion above, using the informal version of McCarthy's McSELF language from the section after this.

$\mathsf{ACK}(m,n) \quad \Leftarrow \quad$ if $m = 0$
then $n+1$
else if $n = 0$
then $\mathsf{ACK}(m-1, 1)$
else $\mathsf{ACK}(m-1, \mathsf{ACK}(m, n-1))$

From this, one can quickly write down the required recursive **ZTEN**-command which computes A:

$$Z \;\equiv\; \mathbf{ite}(x_1 \approx 0)(x_0 \dot{\leftarrow} x_2{+}1)$$
$$(\mathbf{ite}(x_2 \approx 0)(x_1 \dot{\leftarrow} \mathit{pred}(x_1)\ ;\ x_2 \dot{\leftarrow} 1\ ;\ Z)$$
$$(\mathcal{B}_1(x_2 \dot{\leftarrow} \mathit{pred}(x_2)\ ;\ Z)\mathcal{E}\ ;\ x_1 \dot{\leftarrow} \mathit{pred}(x_1)\ ;\ x_2 \dot{\leftarrow} x_0\ ;\ Z))\ .$$

We have used shortcut assignment commands with the predecessor function.

The proof of (ii) in the theorem is basically to show that A grows more quickly than any primitive recursive function. For each $r \geq 0$, define

$$B_r \;:=\; \{\ F : \mathbf{N}^k \to \mathbf{N} \mid k \geq 1 \ \text{ and } \ \forall \vec{a} \in \mathbf{N}^k\ ,\ F(\vec{a}) \leq A(r, \max \vec{a})\ \}\ ,$$

where, for $\vec{a} = (a_1, \cdots, a_k)$, by $\max \vec{a}$, we mean the maximum of $a_1, \cdots, a_k$.

Lemma IV-4.10.2 *If F is primitive recursive, then $F \in B_r$ for some r.*

The statement "$F \in B_r$" is more-or-less synonomous with "F grows no faster than the rth column in the diagram".

Assuming the lemma, the previous fast growing diagonal quickly gives a proof, by contradiction, of (ii) in the theorem :

Suppose that A were primitive recursive. Define $D : \mathbf{N} \to \mathbf{N}$ by

$$D(n) := A(n, n) + 1 \ .$$

Then D is also primitive recursive, being obtained from A by compositions. So $D \in B_r$ for some r. Thus, for all n, we have $D(n) \leq A(r, n)$. Letting $n = r$, we get the contradiction $\quad A(r, r) + 1 \leq A(r, r)$.

Proof of the lemma. First note that, in the diagram, the columns are increasing sequences, since each successive column is a subsequence of the column to its left. And that implies that the rows increase—the entry immediately to the right of a given entry also occurs somewhere in the column above that given entry. Also

$$A(m, n) \geq n + 2 \ , \quad \text{except} \quad A(0, n) = n + 1 \qquad (1)$$

since $A(m, n) \geq A(0, n) + 1$ for $m > 0$. Next, the 'anti-diagonals', $\nwarrow$, weakly decrease, i.e.

$$A(m, n + 1) \leq A(m + 1, n) \qquad (2)$$

We get equality for $n = 0$ by the definition of A. For $n > 0$,

$A(m{+}1, n) = A(m, A(m{+}1, n{-}1)) \geq A(m, n{+}1)$, by (1) and increasing columns.

Finally

$$A(m, 2n) \leq A(m + 2, n) \qquad (3)$$

For $n = 0$, the increasing 0th row gives this. Then, inductively on n,

$$\begin{aligned} A(m + 2, n + 1) \ &= \ A(m + 1, A(m + 2, n)) \geq A(m + 1, A(m, 2n)) \\ &\geq A(m + 1, 2n + 1) \ = \ A(m, A(m + 1, 2n)) \geq A(m, 2n + 2) \ . \end{aligned}$$

The inequalities use the inductive hypothesis and (1) twice, respectively, also using increasing columns in each case.

Now we proceed to establish the lemma by induction on the definition of primitive recursive functions. We'll use the alternative starter functions on page 55.

(i) *The starter functions are in B_0 :*

$$CT_{1,0}(n) = 0 < A(0,n) \;\; ; \;\; \text{so} \;\; CT_{1,0} \in B_0 \; .$$

$$SC(n) = n+1 = A(0,n) \;\; ; \;\; \text{so} \;\; SC \in B_0 \; .$$

$$PJ_{k,i}(\vec{n}) = n_i \leq \max \vec{n} < A(0, \max \vec{n}) \;\; ; \;\; \text{so} \;\; PJ_{k,i} \in B_0 \; .$$

(ii) *If $G, H_1, \cdots, H_\ell$ are all in B_r, and F comes from them by composition, then $F \in B_{r+2}$:* To begin, using (2) for the first inequality,

$$A(r+2,n) \geq A(r+1,n+1) \;=\; A(r, A(r+1,n)) > A(r, A(r,n)) \;\; .$$

The second inequality uses that rows and columns increase. We then get the inequality which shows that $F \in B_{r+2}$:

$$F(\vec{n}) \;=\; G(H_1(\vec{n}), \cdots, H_\ell(\vec{n})) \leq \; A(r \; , \; \max \, (H_1(\vec{n}), \cdots, H_\ell(\vec{n})) \quad \text{since } G \in B_r$$

$$\leq \; A(r \; , \; A(r \; , \; \max \vec{n})) \quad \text{since each } H_i \in B_r$$

$$< \; A(r+2 \; , \; \max \vec{n}) \quad \text{by the previous display.}$$

(iii) *If G and H are in B_r, and F comes from them by primitive recursion, then $F \in B_{r+3}$:* Here we have

$$F(\vec{a}, 0) \;=\; G(\vec{a}) \qquad ; \qquad F(\vec{a}, b+1) \;=\; H(\vec{a}, b, F(\vec{a}, b)) \;\; .$$

First we show by induction on b that

$$F(\vec{a}, b) \;\leq\; A(r+1, b + \max \vec{a}) \qquad\qquad (*)$$

When $b = 0$, $\quad F(\vec{a}, 0) \;=\; G(\vec{a}) \;\leq\; A(r, \max \vec{a}) < A(r+1, \max \vec{a}) \; .$

For the inductive step

$$F(\vec{a}, b{+}1) \;=\; H(\vec{a}, b, F(\vec{a}, b)) \leq A(r\ ,\ \max\{a_1, \cdots, a_k, b, F(\vec{a}, b)\}) \text{ since } H \in B_r$$

$$\leq A(r\ ,\ A(r+1\ ,\ b + \max \vec{a}))$$

(by the inductive hypothesis, because all a_i's and b are less than $A(r+1, b+\max \vec{a})$, and since the rth column increases)

$$= A(r+1, b + \max \vec{a} + 1)\ \text{, as required, by the definition of } A\ .$$

Now we get the required inequality as follows :

$$F(\vec{a}, b) \leq\ A(r{+}1\ ,\ b{+}\max \vec{a}) \ \text{ by } (*)$$

$$\leq\ A(r+1\ ,\ 2\max\{a_1, \cdots, a_k, b\}) \qquad \text{since the } (r+1)\text{st column increases}$$

$$\leq\ A(r+3\ ,\ \max\{a_1, \cdots, a_k, b\}) \qquad \text{by (3)}\ .$$

IV-11: McSELFish Calculations.

Evidently, our 'mechanical' definitions (so far) of the word "computable" are naturally suited to algorithms for processing *sequences* or *strings* of symbols—the computation of *functions* comes as a kind of afterthought. Here we are referring to the string of tape symbols in a Turing machine, and the string of bin numbers in **BTEN**-computability. String computations are more natural in these contexts, and in many CS contexts. The elegant CS text [**J**] emphasizes that point-of-view. See the last few paragraphs of this section for further comment.

The recursive function point of view is the opposite side of the coin. There we only referred vaguely to "recursive verification columns" as giving algorithms for computing the functions. (Of course, getting **BTEN**-commands for these functions was rather easy.) Here we present a very elegant language due to McCarthy, which we call **McSELF**, for giving recursive ('self-referential') algorithms to compute all the (partial) recursive functions. This has several edifying aspects :

(1) It gives a new slant on the recursion/fixed point theorem of Kleene (but is not dependent on it), helping one to see how that theorem is a huge generalization of primitive recursion.

(2) It provides yet another definition of "computability", this time presenting the verbs "to compute" and "to define recursively" as almost synonomous. McCarthy's is perhaps the optimal version of that 'most general recursion scheme' sought by Hilbert et al, to which we referred in the last section, when commenting on the historical significance of Ackermann's function.

(3) It shows how one needs only command-concatenation (function composition) and if-then-else constructions to get a perfectly general definition of computability. At first glance, this seems to preclude infinite loops, and so to contradict what we know to be a required feature of any general computation scheme. But here, that aspect will be generated by the self-referential commands.

To start, we give a few informal examples of recursive procedures, to supplement the one for Ackermann's function in the proof of (i) of **IV-4.10.1**.

$\mathsf{FUN}(n) \quad \Leftarrow \quad$ if $n = 0$
then 0
else $\mathsf{FUN}(n+1)$

The function FUN computed is the 1-variable function which maps 0 to 0, but is undefined on all $n > 0$.

$$\begin{array}{lll} \mathsf{FACT}(n) & \Leftarrow & \text{if } n = 0 \\ & & \text{then } 1 \\ & & \text{else } n \cdot \mathsf{FACT}(n-1) \end{array}$$

The function FACT computed is of course the factorial function $n \mapsto n!$.

$$\begin{array}{lll} \mathsf{PREPRED}(m,n) & \Leftarrow & \text{if } m+1 = n \\ & & \text{then } m \\ & & \text{else } \mathsf{PREPRED}(m+1,n) \end{array}$$

Then

$$\mathsf{PREPRED}(m,n) \;=\; \begin{cases} n-1 & \text{if } m < n \ ; \\ err & \text{if } m \geq n \ . \end{cases}$$

We then get the predecessor function PRED (undefined at 0 and otherwise mapping n to $n-1$) by $\mathsf{PRED}(n) = \mathsf{PREPRED}(0,n)$.

Roughly speaking, the rules of the game are as follows. For starters, we are allowed the zero, successor and projection functions, and the equality function (predicate) $EQ(m,n)$, which is 0 just when $m \neq n$. Then we build procedures by *composing*, and by *if-then-else*, the latter both direct and self-referential/recursive. The 'if-line' would usually be a predicate, but we'll use any procedure already defined for computing a partial function, and say you get "*err*" (i.e. an infinite loop) when the if-line function takes any value other than 0 or 1 (including the value *err*). The 'then-line' and the 'else-line' will be **McSELF**-procedures, already defined in the direct case, but allowed to contain the name of the procedure being defined in the self-referential case. All three lines will use procedures of the same arity (i.e. the same number of variables).

Some presentations of McCarthy's procedures would then continue with this mixed syntax/semantics description, giving rules in great detail for how you operate with self-referential procedures. We shall present it more in a pure mathematical manner, keeping the syntax and semantics at arms-length from each other. The resulting formal system is less attractive to the eye syntactically; but it does allow one to give a crisp version of the semantics—i.e. the definitions of which function a given string of symbols is supposed to compute. Both styles will be used for presenting examples.

The main jobs will then be to

(1) define the semantics;
(2) show that $\mathcal{RC} \Longrightarrow$**McSELF**-computable;
(3) show that **McSELF**-computable $\Longrightarrow \mathcal{BC}$.
By (2) and (3), **McSELF**ish computations constitute a perfectly general computation scheme. For example, they give exactly the Turing machine computable functions. A masochist could also reduce any string computation to this by use of Gödel numbering. But see the last few paragraphs of this section as well.

Definitions. **McELF** is the smallest set of symbol strings which contains atomic strings (symbols)

$$\mathsf{0} \ , \ S \ , \ EQ \ , \ P_{i,k} \quad (1 \le i \le k) \quad \text{of arities } 1 \ , \ 1 \ , \ 2 \ , \ k \text{ respectively,}$$

and is closed under two ways of forming longer from shorter strings :

(1) $\boxed{A \circ (B_1, \cdots, B_\ell)}$ of arity k,with each B_i of arity k, and A of arity ℓ.

(2) $\boxed{\text{if } P \text{ then } Q \text{ else } R}$ of arity k, with each of P, Q and R of arity k.

McZELF is the same, except that we add atomic symbols $Z_1, Z_2, Z_3, \cdots$, where Z_j has arity j, and where the P's occurring in (2) have no Z_j in them (i.e. they are in **McELF**), though the Q's and R's are in **McZELF**.

McSELF consists of all strings from **McZELF**, together with all strings

$$Z_k \equiv A \quad , \quad \text{where } A \text{ is a } \textbf{McZELF} \text{ string of arity } k \ .$$

For each string B from **McSELF** of arity k, and $n_i \in \mathbf{N}$, below we shall define

$$B(\vec{n}) \ = \ B(n_1, \cdots, n_k) \ \in \ \mathbf{N} \cup \{err\} \ ,$$

which gives the semantics. (Function and procedure have the same name.)

$$\mathsf{0}(n) := 0 \ \ ; \ \ S(n) := n+1 \ \ ; \ \ P_{i,k}(\vec{n}) := n_i \ \ ; \ \ Z_k(\vec{n}) := err \ \ ;$$

$$EQ(m,n) \ := \ \begin{cases} 0 & \text{if } m \neq n \ ; \\ 1 & \text{if } m = n \ ; \end{cases}$$

$$A \circ (B_1, \cdots, B_\ell)(\vec{n}) := A(B_1(\vec{n}), \cdots, B_\ell(\vec{n})) \ ;$$

and

$$(\text{if } P \text{ then } Q \text{ else } R)(\vec{n}) \ := \ \begin{cases} Q(\vec{n}) & \text{if } P(\vec{n}) = 1 \ ; \\ R(\vec{n}) & \text{if } P(\vec{n}) = 0 \ ; \\ err & \text{otherwise} \ . \end{cases}$$

The semantics of $Z_k \equiv B$ is more subtle, of course; but it follows exactly the pattern earlier for $Z \equiv C$ where C is a command in **ZTEN**. If B is a **McZELF**-string of arity k, define strings $A_\ell(B)$ for $\ell \geq 0$ by induction on ℓ by

$$A_0(B) \ := \ Z_k \quad ; \quad A_{\ell+1}(B) \ := \ A_\ell(B)^{[Z_k \mapsto B]} \ .$$

As usual, $D^{[Z_k \mapsto B]}$ is obtained by replacing each symbol Z_k in the string D by the string B. It follows easily by induction on **McZELF**-strings that if D is a **McZELF**-string, then so is $D^{[Z_k \mapsto B]}$. The pattern of results in the section on **ZTEN** recursive commands can almost be copied word-for-word here, culminating in the following analogue of **IV-9.5.**

Theorem IV-11.1. *If $A_\ell(B)(\vec{n}) \neq err$, then $\forall \ m \geq \ell$, we have*

$$A_m(B)(\vec{n}) = A_\ell(B)(\vec{n}) \ .$$

Definition.

$$(Z_k \equiv B)(\vec{n}) \ := \ \begin{cases} A_\ell(B)(\vec{n}) \neq err & \text{if such an } \ell \text{ exists} \ ; \\ err & \text{otherwise} \ . \end{cases}$$

The reader may wish to cogitate on these formal definitions to become convinced that the semantics of the self-referential strings, as above, does actually correspond to the rules which you follow intuitively, when working out a particular value, computing with an informal version of such a procedure.

Before going on to show that **McSELF**-computability coincides with **BTEN**-computability, here are a bunch of examples. For the previously given examples, we re-write in the formal style. We shall often write P_{23} , etc., instead of $P_{2,3}$.

FUN : $Z_1 \ \equiv \ \text{if } EQ \circ (P_{11}, \mathsf{0}) \text{ then } \mathsf{0} \text{ else } Z_1 \circ S$

PREPRED : $Z_2 \ \equiv \ \text{if } EQ \circ (S \circ P_{12}, P_{22}) \text{ then } P_{12} \text{ else } Z_2 \circ (S \circ P_{12}, P_{22})$

PRED : $\mathsf{PREPRED} \circ (\mathsf{0}, P_{11})$

FACT : This will be left as an exercise, to be done after MULT below.

ADD, the addition function :

$$\begin{array}{lll} \mathsf{ADD}(m,n) & \Leftarrow & \text{if } n = 0 \\ & & \text{then } m \\ & & \text{else } \mathsf{ADD}(m+1, n-1) \end{array}$$

Formally,

$$Z_2 \equiv \text{if } EQ \circ (P_{22}, \mathsf{0}) \text{ then } P_{12} \text{ else } Z_2 \circ (S \circ P_{12}, PRED \circ P_{22})$$

MULT, the multiplication function :

$$\begin{array}{lll} \mathsf{MULT}(m,n) & \Leftarrow & \text{if } n = 0 \\ & & \text{then } 0 \\ & & \text{else } \mathsf{ADD}(m, MULT(m, n-1)) \end{array}$$

Formally,

$$Z_2 \equiv \text{if } EQ \circ (P_{22}, \mathsf{0}) \text{ then } \mathsf{0} \text{ else } ADD \circ (P_{12}, Z_2 \circ (P_{12}, PRED \circ P_{22}))$$

LEQ, the characteristic function of $\leq$:

$$\begin{array}{lll} \mathsf{LEQ}(m,n) & \Leftarrow & \text{if } m = 0 \\ & & \text{then } 1 \\ & & \text{else} \quad \text{if } n = 0 \\ & & \qquad \text{then } 0 \\ & & \qquad \text{else } \mathsf{LEQ}(m-1, n-1) \end{array}$$

Formally,

$$Z_2 \equiv \text{if } EQ \circ (P_{12}, \mathsf{0} \circ P_{12}) \text{ then } S \circ \mathsf{0} \circ P_{12} \text{ else if } EQ \circ (P_{22}, \mathsf{0} \circ P_{22})$$
$$\text{then } S \circ \mathsf{0} \circ P_{12} \text{ else } Z_2 \circ (PRED \circ P_{12}, PRED \circ P_{22})$$

Given procedures for A and for B, here are two procedures for logical connectives applied to them. These yield the expected predicate when A and B are predicates, but are applicable to any A and B. Other connectives then come for free, of course.

$$\begin{array}{lll} \neg A & \Leftarrow & \text{if } A(\vec{n}) = 0 \\ & & \text{then } 1 \\ & & \text{else } 0 \end{array}$$

Formally : if $EQ \circ (A, \mathsf{0} \circ P_{1,k})$ then $S \circ \mathsf{0} \circ P_{1,k}$ else $\mathsf{0} \circ P_{1,k}$

$A \wedge B \quad \Leftarrow \quad$ if $A(\vec{n}) = 0$
then 0
else if $B(\vec{n}) = 0$
then 0
else 1

Formally : if $EQ \circ (A, \mathsf{0} \circ P_{1,k})$ then $\mathsf{0} \circ P_{1,k}$
else if $EQ \circ (B, \mathsf{0} \circ P_{1,k})$ then $\mathsf{0} \circ P_{1,k}$ else $S \circ \mathsf{0} \circ P_{1,k}$

Notice that the last two use non-self-referential if-then-else strings. Every non-self-referential procedure could be regarded as a 'phoney' self-referential one by just writing "$Z_k \equiv$" in front of it. The semantics is the same if no Z_k occurs after the $\equiv$.

LESS, the characteristic function of $<$, is simply $\mathsf{LEQ} \wedge \neg \mathsf{EQ}$.

See the end of the section for a few interesting and not entirely trivial number theoretic procedures. Here is the formal version of the informal one for Ackermann's function which appeared in the previous section :

$Z_2 \equiv$ if $EQ \circ (P_{12}, \mathsf{0} \circ P_{12})$ then $S \circ P_{22}$ else if $EQ \circ (P_{22}, \mathsf{0} \circ P_{22})$

then $Z_2 \circ (PRED \circ P_{12}, S \circ \mathsf{0} \circ P_{12})$ else $Z_2 \circ (PRED \circ P_{12}, Z_2 \circ (P_{12}, PRED \circ P_{22}))$

As a prelude to the proof that all recursive functions are **McSELF**-computable, note that we have above shown that all the starter functions (in the definition of *recursive function*), namely ADD, MULT, projections, and the characteristic function of $<$ (or its negation), are **McSELF**-computable. And, by definition, the **McSELF**-computables are closed under composition. So it will only remain to show them closed under minimization.

However, it is also interesting to recall that we had an alternative set of starters for the primitive recursive functions, namely $\mathsf{0}$, S, and the projections. So these could be used as starters in defining what (partial) recursive functions are, as long as primitive recursion is added to the other two operations. (As an exercise, the reader should try to prove that, despite this, we cannot dispense with EQ as one of our starters for **McSELF**.) In any case, let's see directly how the **McSELF**-computables are closed under primitive recursion.

So suppose we have F, of arity k, given in terms of G and H :

$$F(\vec{n}, 0) = G(\vec{n}) \quad ; \quad F(\vec{n}, m+1) = H(\vec{n}, m, F(\vec{n}, m)) \quad .$$

An informal procedure for F in terms of ones for G and H is

$$\begin{array}{lll} F(\vec{n},m) & \Leftarrow & \text{if } m=0 \\ & & \text{then } G(\vec{n}) \\ & & \text{else } H(\vec{n},m-1,F(\vec{n},m-1)) \end{array}$$

Formally,

$$Z_k \equiv \text{ if } EQ\circ(P_{k,k},\mathsf{0}\circ P_{k,k}) \text{ then } G\circ(P_{1,k},P_{2,k},\cdots,P_{k-1,k})$$
$$\text{else } H\circ(P_{1,k},\cdots,P_{k-1,k},PRED\circ P_{k,k},Z_k\circ(P_{1,k},\cdots,P_{k-1,k},PRED\circ P_{k,k}))$$

Theorem IV-11.2. *Any recursive function is* **McSELF***-computable.*

Proof. As mentioned above, we need only prove the **McSELF**-computability of the minimization of a **McSELF**-computable function, say G of arity k. Define H, also of arity k, by

$$H(\vec{n},m) := \min\{\ r\ :\ r\geq m \text{ and } G(\vec{n},r)=0\ \}\ .$$

Then $F(\vec{n})$, defined to be $\min\{r : G(\vec{n},r)=0\}$, is just $H(\vec{n},0)$, so it suffices to find a **McSELF**-procedure for computing H . Informally, here it is :

$$\begin{array}{lll} H(\vec{n},m) & \Leftarrow & \text{if } G(\vec{n},m)=0 \\ & & \text{then } m \\ & & \text{else } H(\vec{n},m+1) \end{array}$$

Formally,

$$Z_k \equiv \text{ if } EQ\circ(G,\mathsf{0}\circ P_{k,k}) \text{ then } P_{k,k} \text{ else } Z_k\circ(P_{1,k},\cdots,P_{k-1,k},S\circ P_{k,k})$$

Theorem IV-11.3. *Any* **McSELF***-computable function is* **BTEN***-computable.*

Proof. We extended **BTEN** to **ZTEN**, and then to what I'll call $\mathbf{ZTEN}\cup\mathbf{ZTEN}_{\text{self}}$, where $\mathbf{ZTEN}_{\text{self}}$ consists of strings $Z\equiv C$ for C in **ZTEN**.

Similarly, **McSELF** is $\mathbf{McZELF}\cup\mathbf{McZELF}_{\text{self}}$ where $\mathbf{McZELF}_{\text{self}}$ consists of strings $Z_k\equiv B$ for B of arity k in **McZELF**.

It is now easy to give below a completely explicit function

$$\mathcal{C}\ :\ \mathbf{McSELF}\ \rightarrow\ \mathbf{ZTEN}\cup\mathbf{ZTEN}_{\text{self}}\ .$$

This function has the property that for any $B \in \mathbf{McSELF}$, if B has arity k, then the command $\mathcal{C}(B)$ computes the same "k"-variable function as B. This will prove the theorem, in view of the fact that we showed how to replace any $\mathbf{ZTEN} \cup \mathbf{ZTEN}_{\text{self}}$-command by a $\mathbf{BTEN}$-command which has the same effect.

We define $\mathcal{C}$ by induction on **McSELF**-strings.

$$\mathcal{C}(\mathsf{0}) \;:=\; x_0 \dot{\leftarrow} 0$$

$$\mathcal{C}(S) \;:=\; x_0 \dot{\leftarrow} x_1 + 1$$

$$\mathcal{C}(P_{i,k}) \;:=\; x_0 \dot{\leftarrow} x_i$$

$$\mathcal{C}(Z_k) \;:=\; Z$$

Letting x_m be the variable with largest subscript in $\mathcal{C}(B_i)$ for $1 \leq i \leq \ell$,

$$\begin{aligned}\mathcal{C}(A \circ (B_1, \cdots, B_\ell)) \;:=\; (\; & \mathcal{B}_{1,2,\ldots,m}\, \mathcal{C}(B_1)\, \mathcal{E}\; ;\; x_{m+1} \dot{\leftarrow} x_0\; ; \\ & \mathcal{B}_{1,2,\ldots,m+1}\, \mathcal{C}(B_2)\, \mathcal{E}\; ;\; x_{m+2} \dot{\leftarrow} x_0\; ; \\ & \mathcal{B}_{1,2,\ldots,m+2}\, \mathcal{C}(B_3)\, \mathcal{E}\; ;\; x_{m+3} \dot{\leftarrow} x_0\; ; \\ & \qquad\qquad \bullet \\ & \qquad\qquad \bullet \\ & \qquad\qquad \bullet \\ & \qquad\qquad \bullet \\ & \mathcal{B}_{1,2,\ldots,m+\ell-2}\, \mathcal{C}(B_{\ell-1})\, \mathcal{E}\; ;\; x_{m+j-1} \dot{\leftarrow} x_0\; ; \\ & \mathcal{B}_{m+1,m+2,\ldots,m+\ell-1}\, \mathcal{C}(B_\ell)\, \mathcal{E}\; ;\; x_j \dot{\leftarrow} x_0\; ; \\ x_{\ell-1} \dot{\leftarrow} x_{m+\ell-1} \quad ; \quad & x_{\ell-2} \dot{\leftarrow} x_{m+\ell-2} \quad ; \quad \bullet\ \bullet\ \bullet \quad ; \quad x_1 \dot{\leftarrow} x_{m+1} \quad ; \quad \mathcal{C}(A)\;)\end{aligned}$$

Let x_N be the variable with largest subscript in $\mathcal{C}(P)$. Define

$$\begin{aligned}\mathcal{C}(\text{if } P \text{ then } Q \text{ else } R) \;:=\; & (\mathcal{B}_{1,\cdots,N}\mathcal{C}(P)\mathcal{E}\; ; \\ & \mathsf{ite}(x_0 \approx 1)(\mathcal{C}(Q))(\mathsf{ite}(x_0 \approx 0)(\mathcal{C}(R))(Z)))\;.\end{aligned}$$

At this point, we have defined a function $\mathbf{McZELF} \to \mathbf{ZTEN}$ which will be the restriction of $\mathcal{C}$. It is straightforward, if somewhat tedious, by induction on **McSELF**-strings, to verify that this function has the property 'of preserving the function computed'.

Now to finish the definition of $\mathcal{C}$, let

$$\mathcal{C}(Z_k \equiv B) \quad := \quad Z \equiv \mathcal{C}(B) \ .$$

Basically because the semantics of these 'self-referential strings' are defined formally the same way, again the function $\mathcal{C}$ continues to have the property of preserving the function computed. But here is some necessary detail on this. Another straightforward, this time not tedious, by induction on **McSELF**-strings D, verifies that

$$\mathcal{C}(D^{[Z_k \to B]}) \ = \ \mathcal{C}(D)^{[Z \to \mathcal{C}(B)]}) \ .$$

It follows directly that

$$\mathcal{C}(A_\ell(B)) \ = \ A_\ell(\mathcal{C}(B)) \ .$$

(The two A_ℓ are different, but not much different!) Now the required result about the semantics of the 'self-referential strings' is immediate from the definition of those semantics.

This completes the proof, which is perhaps closer to the spirit of the quote from [**D-W-S**] at the end of the abstract of this paper, than is the only other partial treatment I've seen of essentially the same theorem, in [**Min**].

Let us finish the section with informal **McSELF**-procedures for some interesting number theoretic functions. Formalizing will be left as an (exceedingly mechanical!) exercise.

$\underline{\mathsf{SBT}}$, the subtraction function :

$\mathsf{SBT}(m,n) \quad \Leftarrow \quad$ if $n = 0$
then m
else $\mathsf{SBT}(m-1, n-1)$

This is undefined for $m < n$.

DIV, the divisibility predicate, $m|n$:

$$\begin{array}{lll} \mathsf{DIV}(m,n) & \Leftarrow & \text{if } n = 0 \\ & & \text{then } 1 \\ & & \text{else if } m \leq n \\ & & \quad \text{then } \mathsf{DIV}(m, n-m) \\ & & \quad \text{else } 0 \end{array}$$

PREPRIM :

$$\begin{array}{lll} \mathsf{PREPRIM}(m,n) & \Leftarrow & \text{if } m = n \\ & & \text{then } 1 \\ & & \text{else if } n|m \\ & & \quad \text{then } 0 \\ & & \quad \text{else } \mathsf{PREPRIM}(m, n+1) \end{array}$$

PRIM, the 'is a prime' predicate :

$$\begin{array}{lll} \mathsf{PRIM}(n) & \Leftarrow & \text{if } n < 2 \\ & & \text{then } 0 \\ & & \text{else } \mathsf{PREPRIM}(n, 2) \end{array}$$

This procedure (which isn't self-referential of course) decides whether n is a prime. The reader is left with the exercise of proving this by figuring out what PREPRIM is doing.

It should be noted that we have presented a version of only the *beginning* of McCarthy's formalism, which actually deals with functions of 'just about anything'. This includes functions of *lists* or *strings*, despite what was said in the first paragraph of the subsection about string computations versus function computations. Note that "*Lisp*" referred to below has its name from "list processing" or "list programming".

> *It is our opinion that the recursive function formalism based on conditional expressions presented in this paper is better than the formalisms which have heretofore been used in recursive function theory both for practical and theoretical purposes. First of all, particular functions in which one may be interested are more easily written down and the resulting expressions are briefer and more understandable. This has been observed in the cases we*

have looked at, and there seems to be a fundamental reason why this is so. This is that both the original Church-Kleene formalism and the formalism using the minimalization operation use integer calculations to control the flow of the calculations. That this can be done is noteworthy, but controlling the flow in this way is less natural than using conditional expressions which control the flow directly.

A similar objection applies to basing the theory of computation on Turing machines. Turing machines are not conceptually different from the automatic computers in general use, but they are very poor in their control structure. Any programmer who has also had to write down Turing machines to compute functions will observe that one has to invent a few artifices and that constructing Turing machines is like programming. Of course, most of the theory of computability deals with questions which are not concerned with the particular ways computations are represented. It is sufficient that computable functions be represented somehow by symbolic expressions, e.g. numbers, and that functions computable in terms of given functions be somehow represented by expressions computable in terms of the expressions representing the original functions. However, a practical theory of computation must be applicable to particular algorithms.

. . . . . .

We now discuss the relation between our formalism and computer programming languages. The formalism has been used as the basis for the Lisp programming system for computing with symbolic expressions and has turned out to be quite practical for this kind of calculation. A particular advantage has been that it is easy to write recursive functions that transform programs, and this makes compilers and other program generators easy to write.

. . . . . .

J. McCarthy

IV-12: Frivolity: Clever Electricians' Devices.

This section is definitely not essential to the main purpose (but will not be relegated to smaller print). The **ATEN/BTEN** definition of computability could be criticized as not being close enough to our image of the automatic action, on some array of symbols, of a machine. What follows should ameliorate that type of criticism. On the other hand, no one should take this subsection as any kind of serious discussion of actual physical machines!

Here we give a naive treatment of how one might reduce the carrying out of an **ATEN**-computation to the operation of a few quasi-physical 'devices', hooked up into 'circuits', which carry information around as 'bits'. The bits are modified and re-transmitted by the devices. A 'memory' starts the process by disgorging copies of all its bits into the circuit, and ends the process as well (in the case of a terminating computation—an infinite loop occurrence is almost *visible* in this model). Every so often during the operation of the circuit, the signal re-enters the memory to make a modification. We'll 'prove' that each **ATEN**-command C has at least one corresponding computational circuit, denoted as  . For the first (but not last) time, it must be said that these 'circuits' bear no resemblance at all to what is really done in constructing computers or any other electronic control devices. At the end of the section we make a few comments about how one presumably could construct and connect these components physically, even if there is no motivation to actually carry out that task. One of these descriptions involves electricity, but not in the way any equipment is really built. In particular, no solid state devices are discussed, only very rudimentary electric currents. So the untechnical among us (author included) have no demands made on their imperfect engineering knowledge. Our purpose is merely to make it completely plain that no initiative, free-will, intelligence, ..., need be applied, once the device is switched on, until its output is examined.

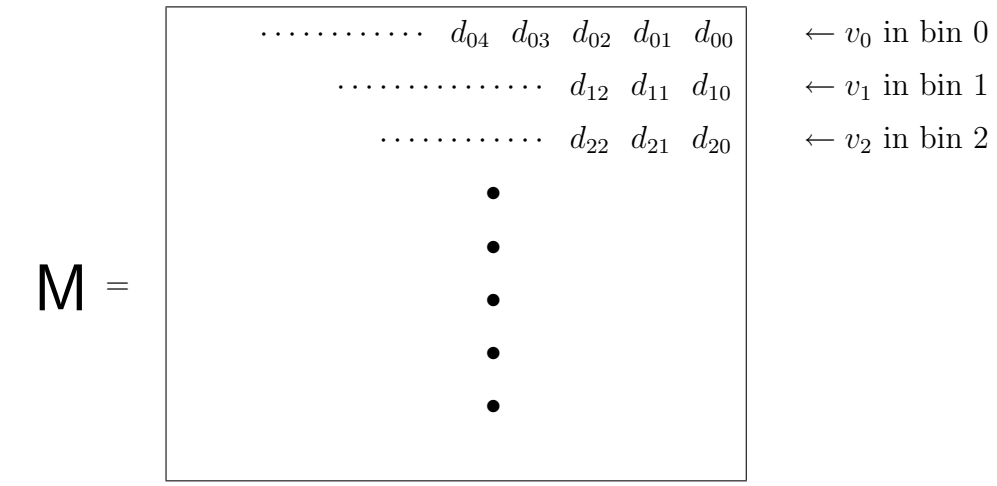
Treatments of "quantum computers" often begin with reference to something called the "circuit model of (classical) computing". This usually won't consist of a purely mathematical definition of *computability*, standing on its own. More likely there will be a vague reference to one, or perhaps to such a definition dependent on the notion of Turing machine. The point of quantum computing is not computability per se, but rather the *efficiency* of algorithms. The so-called circuit model is handy, because, to get a quantum computer, one 'merely' replaces (classical) logic gates by quantum gates.

The transmogrification below of any **ATEN**-computation into pictures involving circuit diagrams is a relatively obvious process. It reinforces our claim that **ATEN**- and **BTEN**-computability do correspond to "mechanical" processes (though that's hardly necessary). Everything is reduced to the *bit* level. But our intention is only that. As we said above, it is *not* to describe how a computer is actually built. Nor is it to give another strictly mathematical definition of computability (though the pictures below could surely be converted into one).

A (classical) **bit** is a *binary digit*, that is, 0 or 1. These will play dual roles, both as digits in the binary representation of natural numbers, and as representing *truth values*, 0 for false and 1 for true. This double role is just a convenience—no philosophical/mystical significance should be attached to it. (The overemphasis by the great pioneering logician George Boole of logic as 'just algebra' seems to occasionally get reversed by modern undergraduates into a regarding of algebra as being 'just logic manipulation'.)

In this section only, a natural number will be identified with an infinite sequence of bits, read from right to left, all but finitely many of these bits being 0. We're thinking of the binary representation, so that, for example, the number whose usual name is 13 will have the name $\cdots 0001101$ in this section.

A **memory** will be an infinite sequence $v_0, v_1, v_2, \cdots$ of numbers, identical with the "bin contents" which we've been dealing with in the other parts of this paper. One might picture the memory device as a rather large square with the latter sequence written downwards, giving an array of bits as follows.

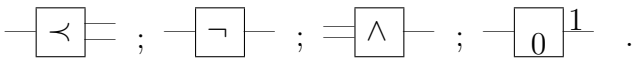


We shall picture three species of **transmission devices**, using one, two or three line segments, usually horizontal. When the segment is partly horizontal with no arrow to indicate direction, the bit, number, or memory is to be *transmitted in the left-to-right direction*.

bit transmission device	**btd**	
number transmission device	**ntd**	
memory transmission device	**mtd**	

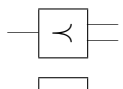
Most people will think of a btd as a single wire. Really, everything comes in bits. And so an ntd is a bundle of btds. And an mtd is a bundle of ntds. In the abstract, ideal, mathematical world, these are infinite bundles. But of course physically they would never be. A physical memory will have a fixed bound both on the size of numbers allowed, and on the maximum quantity of numbers that can be stored.

Now let's go to the bit level and talk about **gates**, often called *logic gates*, though a preferable terminology might be *arithmetic/logic gates*. We shall assume that devices pictured as follows can be constructed in large quantities.

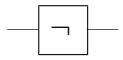


(Again, a warning to readers that although the middle two below are the same as the abstract versions of actual gates constructed out of resistors and transistors, the first is really just a junction and the last has no existence that I know of as a useful engineering device. Furthermore, the standard notation for the middle two in books with actual circuit diagrams containing electronic devices is quite different. Our notation is chosen to make life easy for the reader who has absorbed **[LM]**, or a similar logic text.)

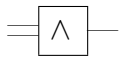
For each of these devices, the input is either one or two bits (coming in from the left), and the output is also either one or two bits.



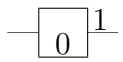
transmits (i.e. "outputs") two copies of the input bit.



transmits the bit opposite to the input bit (e.g. 0 in ⇒ 1 out).



transmits 1 if both input bits are 1; otherwise transmits 0.



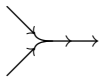
transmits 1 if that's the input bit; otherwise does nothing.

Perhaps this last device overheats a bit, if 0 is the input!

Below we shall use these to construct many other 'gates', eventually several with number or memory input/output, rather than just bits. But first let's list the remaining initial assumptions ('axioms', if you like—but this is a quasi-physical game, not really mathematics). We assume that :

(1) We can attach single btds to each other, in hooking these gates together.

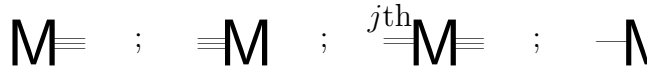
(2) We can splice a pair of btds onto a single btd, which takes the signal from whichever of those two it comes, pictured as

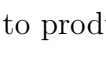


When one of these appears in a circuit, it must be checked that, at a given stage, information can come from at most one of the two directions, not both.

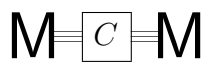
(3) We can split ntds and mtds into two copies as in the first gate above for btds, and also splice ntds and mtds as in (2).

(4) We have a memory device M which takes part in the following pictures:



Ultimately, we shall describe how to produce pictures of the form 

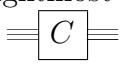
, one for each command C in **ATEN**. The box pictures a 'compound device' for making computations using command C, with the memory coming in from the left on an mtd, and its modified version going out on the mtd to the right. What this means is that if $\underline{v}$ comes in on the left, then $\|C\|(\underline{v})$ goes out on the right (or nothing at all comes out if $\|C\|(\underline{v}) = err$). So perhaps a picture of a special-purpose 'computer' for computing C would be this:



In the second previous display, the two leftmost pictures of the memory will not occur *within* a diagram for producing 

, only as in the imm-

ediately previous display, where the mtd on the left indicates a start button has been pushed, and the one on the right, that a bell has perhaps been rung to wake up the operator so he/she can look at the screen to see the output, and then maybe to go back to sleep again.

In the second previous display, the two rightmost pictures of the memory will occur within diagrams for producing  as follows:

 has the input number being placed as v_j, the jth memory entry. It first erases what had been in the jth bin, and afterwards re-pushes the start button to transmit the new version of memory further down the circuit.

 has the input bit re-pushing to start, not modifying the memory.

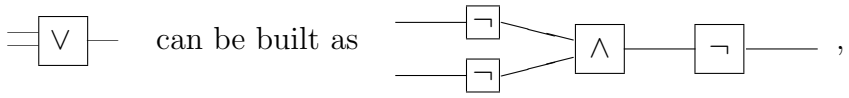
The picture of the 'computer' above (as well as the last few circuits given at the very end of this section) would represent a computer which is 'all-purpose', or 'universal', or 'genuine', when C is a command for computing a universal function $U = U_2$ of two variables. Typically for such C, the input configuration of the memory would have 0's in all bins, except for the Gödel number of the actual command ('software') we wish to compute as v_1, and the Cantor number (CTR) of the input tuple as the number v_2.

Now let's get on with building these circuits.

An elementary theorem of propositional logic is that *the set* $\{ \neg , \ \wedge \}$, *containing only negation and conjunction, is an adequate set of connectives.* (See Theorem 4.2 in [**LM**].) Basically, this just says that, up to truth equivalence, every propositional formula can be built out of those two connectives, and that every possible truth table column **is** the truth table column of at least one formula. One can use the DNF (disjunctive normal form) to prove this, combined with getting rid of the disjunctions, $\vee$, in it by means of the truth equivalence of $F \vee G$ with $\neg(\neg F \wedge \neg G)$.

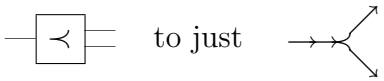
A more 'physical' version of this theorem is this:
Every one-bit-output-gate can be built from the gates given in our initial assumptions above. (The last of the four isn't needed for this.) To be a little more exact: Every function $\{0,1\}^k \to \{0,1\}$ can be implemented by a gate built from the first three gates given in our initial assumptions above. *We now take this as a fact.* But we shall illustrate it with several of the most important examples.

The *"inclusive or" gate*,



using the equivalence above. This gate transmits a 0 if both its inputs are 0, otherwise it transmits a 1.

In circuit diagrams below, we shall simplify the gate



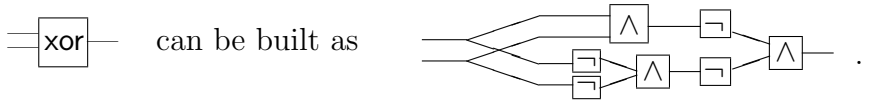
, usually without the arrows.

With double or triple lines, a similar Y-shape indicates duplicating a number or a memory, respectively, along ntds or mtds, respectively.

Using the equivalence

$$P \text{ xor } Q \quad \overset{\text{treq}}{} \quad (P \vee Q) \wedge \neg(P \wedge Q) \quad \overset{\text{treq}}{} \quad \neg(\neg P \wedge \neg Q) \wedge \neg(P \wedge Q) \ ,$$

the *"exclusive or" gate*,

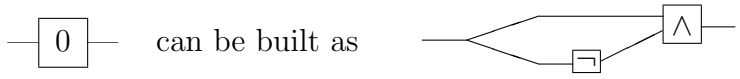


.

This gate transmits a 0 if its inputs agree; it transmits a 1 if they differ.

Two of these gates also play an arithmetical role: the xor-gate implements addition (mod 2) of bits; and the $\wedge$-gate gives the 'carry bit' when adding in binary notation, as well as implementing multiplication (mod 2) of bits.

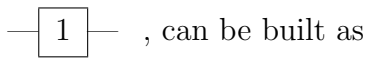
Another use of the splitting gate produces the zero gate:



.

This gate transmits a 0 no matter which bit is its input [formula $P \wedge \neg P$].

The gate which transmits a 1 no matter which bit is its input,



[formula $\neg(P \wedge \neg P)$].

A major use of the splitting gate is to pass from 'scalar outputs' to 'vector outputs'; that is, to implement by gates all functions

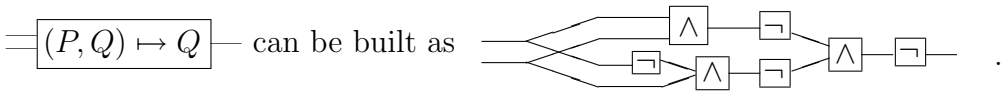
$$\{0,1\}^k \to \{0,1\}^\ell$$

for all k and ℓ. This is clear since we can begin on the left by splitting the entire k-bit input (as done above for the two-bit input to the xor-gate) into a total of "ℓ" streams, and then just do one diagram for each of the "ℓ" component outputs, in the correct order from top-to-bottom. That ordering will be our convention for both input and output in diagrams: reading sequences from left-to-right will correspond to reading both input and output btds in a diagram from top-to-bottom.

Here is another illustration of using the DNF : We have

$$Q \overset{\text{treq}}{} (P \vee \neg P) \wedge Q \overset{\text{treq}}{} (P \wedge Q) \vee (\neg P \wedge Q) \overset{\text{treq}}{} \neg(\neg(P \wedge Q) \wedge \neg(\neg P \wedge Q)) \, .$$

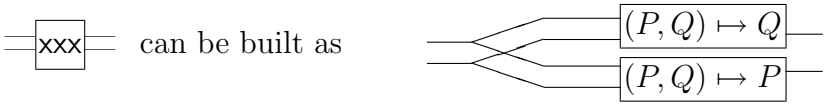
So the gate which simply transmits its *lower* input bit,



.

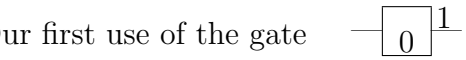
This gate transmits the second of its two inputs, as we said above.

One could draw a diagram for the *upper* input as well, but the theorem guarantees its existence anyway. Also we know there will be a 'cross-over' gate without the need to get explicit, but here it is :



.

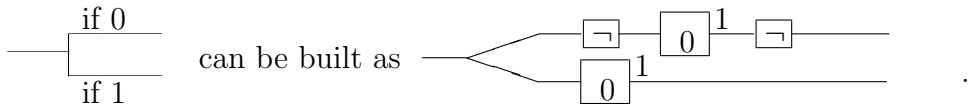
(A practical person can probably think of a simpler way to achieve this in the real world! However, just crossing wires could cause a problem—a gate isn't supposed to transmit anything till *all* its input has arrived.) This gate transmits its 2-bit input in reverse order.

Our first use of the gate 

is for a gate which refuses to transmit anything at all : the *terminator*

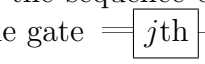
.

A more interesting use is to construct a device which breaks the transmission into two routes, following one route if the input is 0, but following the other if the input is 1 :



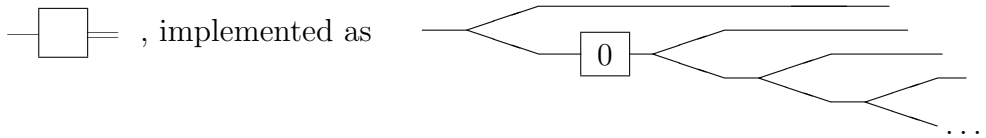
Let us now begin to produce some devices which involve numbers and memory, using ntds and mtds.

Thinking of an ntd as a bundle of btds, we can put a terminator at the end of each of those btds to produce a *terminator for ntds*.

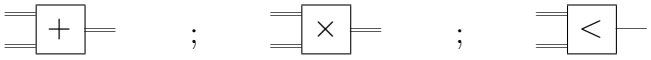
The gate which has a single number input and *transmits the jth digit* is constructed simply by placing a terminator at the end of all but the jth btd in the sequence of btds which makes up that ntd. This will be pictured as the gate 

.
Notice that this assumes that the signal on an ntd consists of a simultaneous sequence of bits transmitted along all its btds. The point being made is that we can't just construct, for example, the 2-bit input gate which transmits the second input by putting a terminator on the upper input. We need a gate for this, as several paragraphs above, which transmits only after receiving a bit from both its input btds.

The gate having a single bit input which it re-transmits as a number is

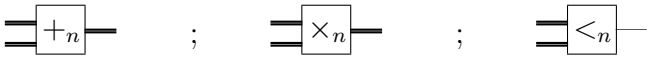


Now we need devices that do arithmetic and inequalities with numbers, not merely with bits. These are three important gates



.

They implement the structure on **N** which the language of 1st order number theory formalizes. In reality, we can only produce, for reasonably large $n \geq 1$, gates

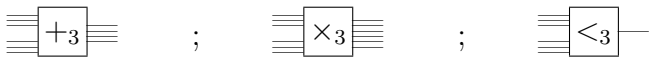


.

where each of the six thick lines giving the three inputs is really a sequence of "n" btds. The thick lines on the two outputs are "$n+1$" btds for $+_n$

and "$2n$" btds for $\times_n$. This reflects the fact that we can treat the sum of a pair of "n"-digit numbers as an "$n+1$"-digit number, and their product as a "$2n$"-digit number.

For example,

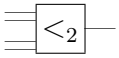


correspond to functions from $\{0,1\}^6$ to $\{0,1\}^4$, $\{0,1\}^6$, and $\{0,1\}$, respectively. We'll only go part ways towards writing these out explicitly, since we do know from theory that it can be done, and that's the point of this section (not to give an instruction manual for building a very inefficient 'computer', which would never work in any practical sense!)

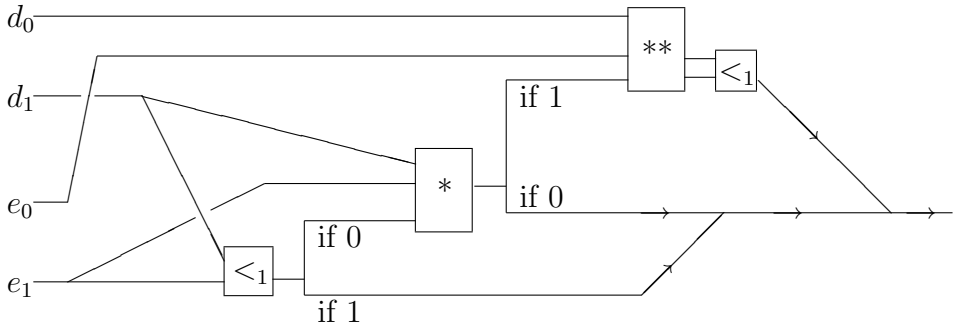
As for the $<$ -relation, first note that

.

This transmits 1 only if the input is $(0,1)$; that is, the upper bit is 0 and the lower is 1.

Then one may obtain 

as follows :



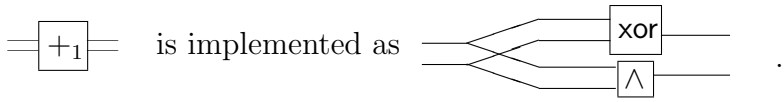
where $*$ maps (1,0,0) to 0, and all other $(x,y,0)$ to 1, but everything else is indifferent; and $**$ maps (x,y,z) to (x,y). We need these two gates in the diagram to ensure that, for any given input, only one path on the right (with arrow) carries an output. Building circuits for $*$ and $**$ is quite easy.

This diagram is just saying, for bits d_0, d_1, e_0, e_1, that

$$d_0+2d_1 < e_0+2e_1 \iff d_1 < e_1 \text{ or } [d_1 \geq e_1 \text{ and } (d_1, e_1) \neq (1,0) \text{ and } d_0 < e_0] \,.$$

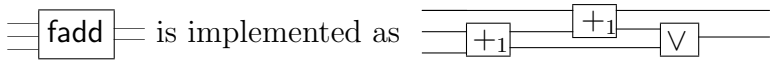
Essentially the same diagram can be used inductively to give a circuit for $<_{n+1}$, in terms of one for $<_n$. Just change the rightmost $<_1$ to $<_n$, change (d_1, e_1) to (d_n, e_n), and change d_0 and e_0 to the strings $d_{n-1}d_{n-2}\cdots d_1 d_0$ and $e_{n-1}e_{n-2}\cdots e_1 e_0$, respectively. The two btds in the diagram for the latter inputs must be changed to two bundles, each with "n" btds.

Now let's deal with addition gates. Using the facts mentioned earlier about getting addition (mod 2) and the "carry digit" from the gates for xor and $\wedge$, respectively, we can have a gate labelled $+_1$ which produces the (mod 2) sum on the upper btd, and the carry digit on the lower one; i.e.

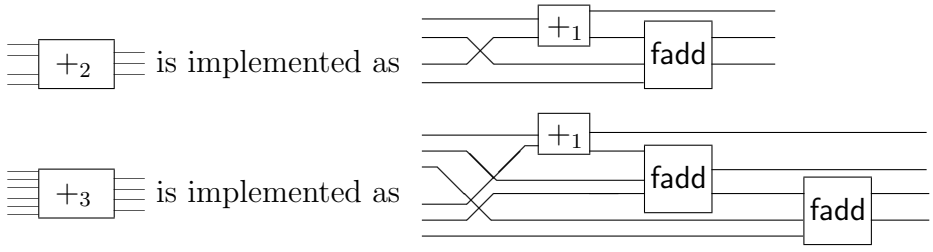


.

Now get a 'full adder' gate :



Its upper output is the (mod 2) sum of the three input bits. The lower one is 0 or 1 according as the three add to at most 1 or at least 2 (a kind of triple carry digit). Now it's not hard to indicate the pattern for producing the gates for $+_n$:



In general, the $+_n$-gate will use "$n-1$" fadd-gates, and a single $+_1$-gate.

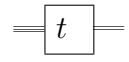
Getting from multiplying bits to multiplying numbers is fairly straightforward now. The usual grade school algorithm, transposed to base 2, is even simpler than in base 10, since the product of two digits is a digit here. First

produce a gate to implement the function $+_{k,\ell} : \{0,1\}^{k\ell} \to \{0,1\}^{k+\ell-1}$ which is the addition of "ℓ" numbers, each composed of "k" digits. Now, supplementing with 0's on the left and right, the usual array from grade school, gotten by multiplying each of the "n" digits of the second number into the first number, corresponds to an easily constructed gate for a function $\{0,1\}^{2n} \to \{0,1\}^{(2n-1)n}$ in which the supplementation produces "$n-1$" blocks of "n" 0's separating the "n" different "n"-digit numbers from these multiplications. Then the gate for $\times_n$ is the succession of three gates realizing the three-fold composition

$$\{0,1\}^{2n} \to \{0,1\}^{(2n-1)n} \to \{0,1\}^{(2n-1)+n-1} \to \{0,1\}^{2n} \quad ,$$

where the third map just drops the last "$n-2$" digits (which will always be zero), and the middle map is $+_{2n-1,n}$. Gates for all three will be left for the reader to construct. Of course, the theorem using the DNF does guarantee that a gate for $\times_n$ exists anyway.

All the basic ingredients are assembled, so now we can inductively produce circuits corresponding to $1^{\underline{\text{st}}}$*order terms*, then $1^{\underline{\text{st}}}$*order (quantifier free) formulas* and finally **ATEN**-*commands*, basing them on the inductive definitions of these concepts.

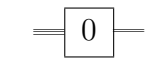
First we need, for every term t, a circuit 

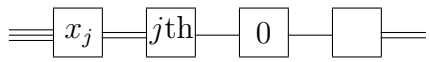
. It will take the memory $\underline{v}$ as input and transmit the number corresponding to the formal string t. This number is denoted $t^{\underline{v}}$ in [**LM**].

When $t = x_j$, the device 

is obtained in the same manner as before (when we 'picked off' the jth digit in a number). Use terminators on all ntds except the jth, in the bundle that makes up the mtd, to pick off the jth number in the memory.

When $t = 0$, for any choice of j,

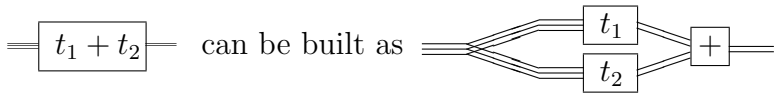


can be built as 

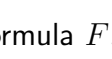
.

When $t = 1$, the construction is the same.

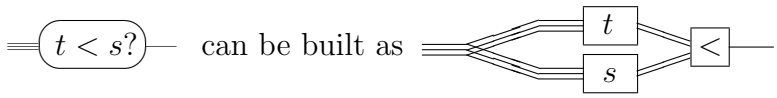
That takes care of atomic terms. Then the inductive steps go as follows.



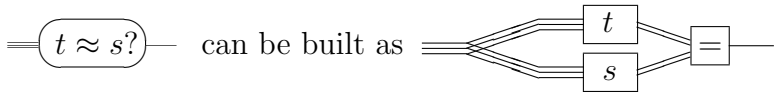
For $t_1 \times t_2$, essentially the same thing works, completing the job.

Next we need, for every quantifier-free formula F, a circuit 

.
It will take the memory $\underline{v}$ as input, and transmit the bit 1 if F is true at $\underline{v}$, but transmit the bit 0 if F is false at $\underline{v}$.

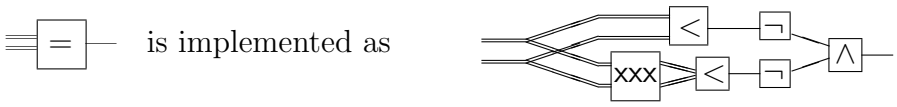


.



,

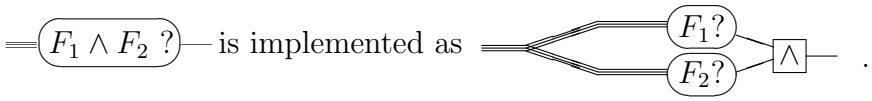
except that we need to provide an 'equality decider', using our 'inequality decider', as follows:



.

Now the inductive steps are easy:

.



.

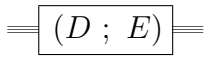
Finally we need, for every command C in **ATEN**, a circuit 

.
It will take the memory $\underline{v}$ as input, and transmit the memory $\|C\|(\underline{v})$, as long as the latter is not err. This is what we have been aiming for.

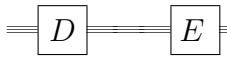
As for atomic commands,

.

Then, proceeding inductively, first note that

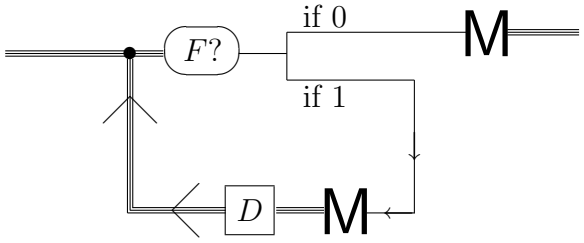


is clearly



.

The final step is to produce a circuit diagram which implements the command **whdo**$(F)(D)$, inductively assuming such a diagram to exist for the command D. We've used arrows in the following diagram wherever the information doesn't flow from left-to-right.



That finishes the real job of this section. The final paragraphs consist of remarks about

(1) how to use these diagrams to recast mathematical definitions from the very early part of the paper in more picturesque form; and

(2) one way to imagine building some kind of quaint 'computer' like this as an actual physical object.

One could use the above to define *computation*, and thereby "computability". For example, suppose we have an **ATEN**-command C and a corresponding circuit diagram 

. This diagram is assumed to be broken down into its smallest components, except that the gates for $+$, $\times$ and $<$ are not broken down further, since they can't really be for our purpose here. For example, no $+_n$ handles more than finitely many numbers, and so we imagine $+$ as being the mythical $+_\infty$, since it's "computability in principle" we are after here. But see the last few paragraphs below.

Now a computation with C using input $\underline{v}$ could be defined to be a sequence of diagrams each of whose terms is the detailed version of the circuit from the paragraph above, embellished with tags attached to some of its btds, ntds and mtds. (Only the tags change, as one goes from one member of the sequence to the next.) The first member of the sequence just has one tag with

$\underline{v}$ written on it, and that tag is attached to the very first mtd leading into the circuit. The remaining members of the sequence are defined by obvious rules of the following form: use the same tags as the previous member of the sequence, except when tags are attached to **all** the tds leading in to ['giving input into'] some gate in the diagram. (We regard M's as "gates" in this context.) All such tags are removed, and new ones attached to all the tds leading out of that gate ['giving the transmitted bit(s) or number(s) or memory' using the obvious formulas realizing the semantics of **ATEN**]. Now possibly the sequence is infinite, in which case it is more suitable to refer to that sequence as an *infinite loop* than as a *computation*. The sequence will be finite (and will end) exactly when we get to a tagged diagram with the tag on the final mtd leading out of the circuit. Formalizing all this is easy and will be left to the reader. At most one additional remark will be necessary : When labelling a tag on the mtd leading away from an M in the diagram, you of course use the memory from the (provably) unique most recent predecessor of that sequence term for which there is a tagged mtd, except for possibly modifying one number in that memory.

What about the crazy idea of imagining the use of such 'circuit diagrams' to actually construct a physical computing device? First of all, we would need to go back entirely to bits, and begin with making a choice as to how many digits, say N, will be allowed for the largest number being represented, and also how many numbers the memory is able to hold. We'll use $+_n, \times_n$ and $<_n$ in place of $+$, $\times$ and $<$, for various $n < N$. And let's ignore 'overflow' questions. Then we know how many btds to use in replacing each ntd and mtd.

It seems like electrical devices are the only natural choice here. But I want to avoid as much as possible any need for technical knowledge (having very little myself), so, for example, no transistors, vacuum tubes, etc. will be mentioned. We need only describe the four basic gates as physical objects, and also the four pictures involving the memory. With some consultation, I have become convinced that the following crude devices could actually be constructed (but shouldn't be!).

Rather than representing bits as 'roughly 5 volts for the 1-bit' and 'roughly 0 volts for the 0-bit', as they apparently exist in the processor of an actual computer, we shall take 'roughly –12 volts for the 1-bit' and 'roughly +12 volts for the 0-bit', as they are often described in texts on the longer-distance

transmission of information, for example, computer to printer. As an option here, we need to have 'no current at all (0 volts) for no information'. Here the voltages are with respect to 'ground'.

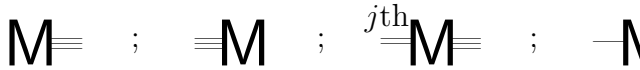
Now our four basic 'gates' will actually have a *pair* of wires for each btd pictured, one to hold the input current or output current, and one to transmit a pulse backwards to turn that current off after the information has been processed by the gate which the first wire leads into.

So, for example, when –12 volts is detected by the ¬-gate from its input wire, it 'arranges' for two things to happen: a pulse goes backwards to the source to shut that –12 volt current off (somehow—no discussion of how here!); and a +12 volt current is (somehow) made to flow in that gate's output wire. Of course, interchanging + and – signs gives the other possible behaviour.

In the case of the ∧-gate, nothing happens till *both* input wires have current, at which point both are shut off, and the appropriate current is induced in its output wire. Our 'curious gate' which permits a 1-bit past, but not a 0-bit, will, for any input current, shut it off; but only for an input of –12 volts will it induce the same in its output wire, not for a +12 volt input. This is where we need the option mentioned above of "no information". If there's any reason to believe that such devices would in principle be impossible to build, I would be grateful (but perhaps chagrined!) to hear about that from any technically expert reader.

Now for the memory pictures. First of all we see this memory device in a fairly standard way as having a speck of some metal at each memory location, magnetized if the 1-bit is there, and unmagnetized if the 0-bit is there. (There is always some bit there.) Of course it is assumed that this memory can be prepared prior to the computation with bits however we wish, as the input to the computation.

Clearly we don't want to have a separate physical object for each picture of M in the 'circuit' corresponding to an **ATEN**-command. So there will be many wires coming in, and many 'going out', at each memory location, roughly one for each occurrence of the memory in the diagram. Each such wire will also have a mate to transmit a pulse back to shut off the current, as with the gates above. So, each picture



needs some discussion.

The first one just indicates one bunch of output wires (plus mates as above), one such pair at each memory location, which, when the entire circuit is switched on, are given a current of either –12 volts or +12 volts, according as the location is magnetized or not.

The second picture indicates one bunch of input wires (plus mates), which, when all have current, will each magnetize, if necessary, the location where it ends, or de-magnetize it (if necessary), according as that input current is –12 or +12 volts. And also, the mates will send a pulse back to shut off those currents (though that's only relevant to a subsequent computation with the same circuit). And I suppose, a bell will be rung to alert the operator, as discussed when these pictures were introduced.

The third picture indicates a bunch of input wires coming into *only* the locations which are assigned for the bits in the binary representation of v_j (and also output wires leading from *all* the locations in the memory). The input currents magnetize or de-magnetize the locations as appropriate, and the mated wires then signal to shut the incoming current off. But these actions take place only after *all* the input wires have current. Also, outgoing currents are induced as with the first picture.

Finally, the 4th picture indicates an input wire at some 'neutral' place in the memory. The result of a current of either positive or negative voltage here is simply to induce an appropriate current in the outgoing wires from all memory locations, and to shut that incoming current off. No alteration to the memory is made.

So, by attaching output wire(s) of one device to input wire(s) of the next, using the 'plan' given by the circuit for the command C, you've got this mythical beast which is supposed to compute with algorithm C, a beast which no one *would* ever build, but which I trust no one will claim that no one (intelligent being of sufficient longevity) *could* ever build! By shutting off the input current(s) each time one of the devices 'does its thing', the looping in the **whdo**-circuits will not cause any instability. Furthermore, the devices with more than one incoming wire will not mistake a leftover current from an earlier part of the computation for new input, and then err, 'doing their thing' prematurely. We're assuming that the shutoff never takes place later than the inducing of the outgoing current(s) on the other side of the device.

So actually, currents in the wires in this picture will correspond exactly to the "tags" used in the more abstract description several paragraphs back, of

a computation taking place. Each successive instance of this can be regarded as a single step in the computation.

But I doubt whether building such a device is of any value, except possibly for actors playing scientists stranded on a desert island in the canonical BBC series. And then I can't imagine what would replace the electric components other than maybe little men in green coats carrying placards, each with a bit written on it, scuttling along pathways which replace the wires. But then the whole point of there being no initiative, nor free-will, nor intelligence needed, to carry on a computation, is slightly obscured, however compliant these little green men might be!

V: Proof of the "Gödel express" .

We shall prove that [$\mathcal{RC} \implies$ *expressible*] for relations (which are regarded as total functions that take no values other than 0 and 1) by proving that [$\mathcal{RC} \implies$ *representable*] for arbitrary (partial) recursive functions. The reason is to permit a proof by induction on recursive functions. Of course, representability must be defined, and proved equivalent to expressibility for relations. This equivalence holds for arbitrary (i.e. possibly partial and non-total) '01-functions'.

After some preliminaries, the definitions of the italicized words above are given in Subsection V-1 at the beginning, and the theorems are stated at the end. Subsections V-2 and V-3 are technically interesting, but not essential to the purpose of this paper. In V-2, we fill in some points left out of the text [**LM**]. In V-3, there is some explanation of the (often mysteriously formulated) assertion that any good proof system for $1^{\underline{\text{st}}}$order logic may be alternatively regarded as being another possible definition of the word *computable*, equivalent to the four discussed in Section II.

First, here is some discussion of the overall organization. Our concern is with a particular set, $\mathcal{N}$, of nine sentences from the language of $1^{\underline{\text{st}}}$order number theory. [If you're impatient to see $\mathcal{N}$, consult either the appendix to V-1, or the beginning of Appendix L in [**LM**] .] More specifically, we want to establish the existence of various derivations from that set. We shall use the proof system for $\vdash$, as opposed to the one for $\vdash^*$, given in [**LM**], whenever any non-sentence formula appears as a premiss in what follows.

Just below is a list of needed results which specifically use the nine sentences from $\mathcal{N}$ in their derivations. Listing $\mathcal{N}$ and proving these results are left to an appendix to V-1. This should help to clear up the somewhat tortuous path of the overall proof in V-1. In particular, it has the advantage of exhibiting just how much 'truth' about the natural numbers is needed for the Gödel express (actually not much). This 'truth' is often called *formal arithmetic*, basically being [*Peano arithmetic with* $1^{\underline{\text{st}}}$*order induction removed*]. We could shorten the proof by *assuming* all the facts about $\mathcal{N}$ listed below as **Formal Arithmetic** (i.e. by expanding $\mathcal{N}$ rather drastically), but it seems better to show how these follow from a *finite* list, $\mathcal{N}$, of premisses. Furthermore, the Gödel express, and the theorems of Church, Gödel and Tarski were formulated that way in [**LM**]. These facts about derivations are very elementary.

The following notation will be used. Firstly, as done in Appendix L from **[LM]**, in our $1^{\underline{\text{st}}}$order number theory language there is a *defined* 1-ary function symbol S , which stands for "successor". It is defined by saying that St is an abbreviation of $(t+1)$, for any term t, including ones which might already contain S. We shall be very fussy distinguishing syntax from semantics, so that $1_{\mathbf{N}}$ and $0_{\mathbf{N}}$ will be used for the actual natural numbers, and $+_{\mathbf{N}}$ for the operation. Fortunately we can just use juxtaposition for actual multiplication, and use $>$ and $\geq$ for comparing numbers, reserving $<$ as a symbol in the formal language. Many will find this overly fussy. But this writeup is a pedagogical effort (or at least an effort to explain this stuff to myself!), not an effort to impress the cognoscenti.

For a natural number k, the fixed term "k" is defined to be $SS\cdots S0$, where the number of copies of S used is the natural number whose name is k ; or, if you prefer, inductively, "$0_{\mathbf{N}}$" $:= 0$ and "$k+_{\mathbf{N}}1_{\mathbf{N}}$" $:= S$"k" $=$ "k"$+1$.

Formal Arithmetic. *As proved in the appendix to this section, the formulae* (I) *to* (VII) *below are all in* $\mathcal{N}^{\vdash}$. *That is, each is derivable from* $\mathcal{N}$, *itself a list of nine specific formulae, also given in the appendix.*

$$\text{(I)} \qquad \text{"}a\text{"} + \text{"}b\text{"} \;\approx\; \text{"}a +_{\mathbf{N}} b\text{"} \qquad \textit{for any} \;\; a, b \in \mathbf{N} \;\; .$$

$$\text{(II)} \qquad \text{"}a\text{"} \times \text{"}b\text{"} \;\approx\; \text{"}ab\text{"} \qquad \textit{for any} \;\; a, b \in \mathbf{N} \;\; .$$

$$\text{(III)} \qquad \neg\ \text{"}a\text{"} \;\approx\; \text{"}b\text{"} \qquad \textit{for any} \;\; a \neq b \text{ in } \mathbf{N} \;\; .$$

$$\text{(IV)} \qquad t < s \;\vee\; t \approx s \;\vee\; s < t \qquad \textit{for any terms} \;\; s \textit{ and } t \;\; .$$

$$\text{(V)} \qquad \text{"}a\text{"} \;<\; \text{"}b\text{"} \qquad \textit{for any} \;\; a, b \in \mathbf{N} \;\; \textit{with} \;\; b > a \;\; .$$

$$\text{(VI)} \qquad \neg\ \text{"}a\text{"} \;<\; \text{"}b\text{"} \qquad \textit{for any} \;\; a, b \in \mathbf{N} \;\; \textit{with} \;\; a \geq b \;\; .$$

$$\text{(VII)} \qquad x < \text{"}\ell\text{"} \;\rightarrow\; J \qquad \textit{for any formula } J, \textit{ any variable } x,$$
$$\textit{and for any} \;\; \ell \in \mathbf{N} \;\; \textit{for which} \;\; \mathcal{N} \vdash J^{[x \mapsto \text{"}i\text{"}]}$$
$$\textit{for each natural number } i \textit{ with } \ell > i \geq 0_{\mathbf{N}} \;\; .$$

As in the previous section, the notation $\vec{v}$ will be short for $(v_1, \cdots, v_k)$, a member of $\mathbf{N}^k$, to distinguish it from the 'infinite vector' $\underline{v} = (v_0, v_1, v_2, \cdots)$. This will be handy for shortening replacement notation, viz. $F^{[\vec{x} \mapsto \text{"}\ \vec{v}\ \text{"}]}$ will be

an abbreviation of $F^{[x_1\to\text{“}v_1\text{”}][x_2\to\text{“}v_2\text{”}]\cdots[x_k\to\text{“}v_k\text{”}]}$, although even the latter is already a substantial improvement on the not uncommon $F_{x_1,\cdots,x_k}[0^{v_1},\cdots,0^{v_k}]$, both in terms of suggestiveness and of readability. (Because each "v" is a *fixed* term, i.e. no variables, the formula $F^{[\vec{x}\to\text{“ }\vec{v}\text{ ”}]}$ can be regarded as being either a *simultaneous* or a *successive* substitution.)

V-1. Representability and Expressibility.

First here are the basic definitions.

A 01-function is a function $f : D \to \mathbf{N}$, where $D \subset \mathbf{N}^k$ for some positive k, such that $f(D) \subset \{0_\mathbf{N}, 1_\mathbf{N}\}$.

A k-ary relation will be regarded as a 01-function $f : \mathbf{N}^k \to \mathbf{N}$, [with $f^{-1}(0_\mathbf{N})$ being the set of k-tuples which are 'related', i.e. $0_\mathbf{N}$ is the 'true' value. That is the opposite convention to the previous sections, but is immaterial to the real issues].

If f is a 01-function with domain D, we say that f is expressible if and only if there is a formula F (in the language of 1st order number theory) such that, for all $\vec{v} \in D$, we have

$$f(\vec{v}) = 0_\mathbf{N} \implies \mathcal{N} \vdash F^{[\vec{x}\to\text{“ }\vec{v}\text{ ”}]}\ , \quad \text{and}$$
$$f(\vec{v}) = 1_\mathbf{N} \implies \mathcal{N} \vdash \neg F^{[\vec{x}\to\text{“ }\vec{v}\text{ ”}]}\ .$$

If g is an arbitrary function $D \to \mathbf{N}$, where D is a subset of some $\mathbf{N}^k$, we say that g is representable if and only if there is a formula G such that, for all $\vec{v} \in D$, we have

$$\mathcal{N} \vdash (x_0 \approx \text{“}g(\vec{v})\text{”} \leftrightarrow G^{[\vec{x}\to\text{“ }\vec{v}\text{ ”}]})\ .$$

Examples.

(A) The 'less-than' relation on $\mathbf{N} \times \mathbf{N}$ corresponds to the 01-function

$$\overline{\chi}_< \ :\ (v_1, v_2) \mapsto \begin{cases} 0_\mathbf{N} & \text{if } v_2 > v_1\ ; \\ 1_\mathbf{N} & \text{if } v_1 \geq v_2\ . \end{cases}$$

I claim that $\overline{\chi}_<$ is expressed by $F \ = \ x_1 < x_2$, (which seems eminently appropriate!) The definition of expressibility reduces in this case to showing, for all $(v_1, v_2) \in \mathbf{N} \times \mathbf{N}$, that

$$v_2 > v_1 \implies \mathcal{N} \vdash \text{“}v_1\text{”} < \text{“}v_2\text{”} \quad \text{and}$$
$$v_1 \geq v_2 \implies \mathcal{N} \vdash \neg\text{“}v_1\text{”} < \text{“}v_2\text{”} \quad .$$

These are (V) and (VI) in the previous list of $\mathcal{N}$-specific derivations.

(B) Using the proof several pages ahead of the connection between expressibility and representability, the example above would produce the fact that the function $\overline{\chi}_<$ is represented by

$$G \;=\; (x_1 < x_2 \wedge x_0 \approx 0) \vee (\neg x_1 < x_2 \wedge x_0 \approx 1) \quad .$$

To directly verify this requires that

$$\mathcal{N} \;\vdash\; (\; x_0 \approx \text{“}\overline{\chi}_<(v_1, v_2)\text{”} \;\leftrightarrow\; G^{[x_1 \to \text{“}v_1\text{”}][x_2 \to \text{“}v_2\text{”}]} \;) \; .$$

That amounts to

$$v_2 > v_1 \implies \mathcal{N} \vdash (x_0 \approx 0 \leftrightarrow (\text{“}v_1\text{”} < \text{“}v_2\text{”} \wedge x_0 \approx 0) \vee (\neg\text{“}v_1\text{”} < \text{“}v_2\text{”} \wedge x_0 \approx 1))$$

and

$$v_1 \geq v_2 \implies \mathcal{N} \vdash (x_0 \approx 1 \leftrightarrow (\text{“}v_1\text{”} < \text{“}v_2\text{”} \wedge x_0 \approx 0) \vee (\neg\text{“}v_1\text{”} < \text{“}v_2\text{”} \wedge x_0 \approx 1)) \; .$$

We won't repeat the general proof below for this special case, and nothing simpler seems to work. Again using that proof, it is entertaining that $\overline{\chi}_<$ is also expressed by $\quad (x_1 < x_2 \wedge 0 \approx 0) \vee (\neg x_1 < x_2 \wedge 0 \approx 1) \quad .$

(C) Suppose that t is a term (in 1^{st}order number theory) in which no variables other than possibly $x_1, \cdots x_k$ appear. Then $t^{\underline{v}}$ depends only on $(v_1, \cdots v_k) = \vec{v}$. The 'really-obviously-$2-hand-calculator-computable' functions are the functions $(v_1, \cdots v_k) = \vec{v} \;\mapsto\; t^{\underline{v}}$ for the various such t and k. These are total functions. Evidently, the most obvious way to represent such a function would be to take the formula G to be $x_0 \approx t$.

To verify this, we need, for all $\vec{v} \in \mathbf{N}^k$, to see that

$$\mathcal{N} \;\vdash\; (x_0 \approx \text{“}t^{\underline{v}}\text{”} \leftrightarrow (x_0 \approx t)^{[\vec{x} \to \text{“ } \vec{v} \text{ ”}]} \;) \; .$$

However $(x_0 \approx t)^{[\vec{x} \to \text{“ } \vec{v} \text{ ”}]}$ is the same as $x_0 \;\approx\; t^{[\vec{x} \to \text{“ } \vec{v} \text{ ”}]}$. When t is as simple as the example $t = x_i$, which gives the projection function, then “$t^{\underline{v}}$”

is actually the same as $t^{[\vec{x}\to\text{“ }\vec{v}\text{ ”}]}$, so there is not much left to do, as $\vdash F \leftrightarrow F$ is clear.

But, in quite simple cases like $t = x_1 + x_2$, which gives the addition function, we need to actually prove something using $\mathcal{N}$, namely to show that there is a derivation of

$$x_0 \approx \text{“}v_1 +_{\mathbf{N}} v_2\text{”} \;\leftrightarrow\; x_0 \approx \text{“}v_1\text{”} + \text{“}v_2\text{”} \ .$$

This is clear from (I) in the previous list of $\mathcal{N}$-specific derivations, using the fact that

$$\Gamma \vdash t \approx s \quad \Rightarrow \quad \Gamma \vdash x_0 \approx s \leftrightarrow x_0 \approx t \ .$$

The situation for the multiplication function is exactly similar.

But the general case is also quite easy—it is almost immediate from (I) and (II) in that previous list, using induction on terms, that

$$\mathcal{N} \;\vdash\; x_0 \approx \text{“}t^{\underline{v}}\text{”} \;\leftrightarrow\; x_0 \approx t^{[\vec{x}\to\text{“ }\vec{v}\text{ ”}]} \ ,$$

(for terms t involving only $x_1, \cdots, x_k$), as required.

Proof by 'induction on recursive functions' that [$\mathcal{RC} \Longrightarrow$ *representable*.]

Example (A)/(B) and the special cases mentioned in (C) take care of the four initial cases (AKA "starter functions"). We are left with the two inductive steps:

Composition:

Suppose that $H_1, \cdots, H_k$ are formulae representing recursive functions $h_1, \cdots, h_k$ respectively, each being a function of n variables. So, for i in the range $k \geq i \geq 1$, and for all $\vec{v} \in D_i = \text{Domain}(h_i) \subset \mathbf{N}^n$, where we use the notation $\vec{v} = (v_1, \cdots, v_n)$ and $\vec{x} = (x_1, \cdots, x_n)$, we have

$$\mathcal{N} \;\vdash\; (x_0 \approx \text{“}h_i(\vec{v})\text{”} \leftrightarrow H_i^{[\vec{x}\to\text{“ }\vec{v}\text{ ”}]}) \ . \qquad (*)_i$$

Also let F represent f, a function of k variables, so we have

$$\mathcal{N} \;\vdash\; (x_0 \approx \text{“}f(w_*)\text{”} \leftrightarrow F^{[x_*\to\text{“}w_*\text{”}]}) \ , \qquad (**)$$

using the notation $w_* = (w_1, \cdots, w_k)$ and $x_* = (x_1, \cdots, x_k)$. This holds for all $w_* \in D$, the domain of f. Because $\Gamma \;\vdash\; G \leftrightarrow H$ implies that

$\Gamma \vdash (G \leftrightarrow H)^{[x_{\overrightarrow{j}} \to 0]}$, we may replace all free occurrences in F of x_j for $j > k$ by 0, and assume to begin with that at most $x_0, x_1, \cdots, x_k$ occur freely in F.

Choose a list of pairwise distinct variables $y_1, \cdots, y_k$, none of which occur in any of F or $H_1, \cdots, H_k$. Define $G_i := H_i^{[x_{\overrightarrow{0}} \to y_i]}$. Substituting y_i for x_i for all i with $k \geq i \geq 1_{\mathbf{N}}$, define $G := F^{[x_{*\overrightarrow{}} \to y_*]}$. Note that we may choose the y_i to be x_j's with $j > n$, so that none of the variables $x_1, \cdots, x_n$ occur free in G.

Then we'll show that the formula

$$\exists y_1 \exists y_2 \cdots \exists y_k (G_1 \wedge \cdots \wedge G_k \wedge G)$$

represents the composition, g, of f with the h_i. In other words, we have

$$g(v_1, \cdots, v_n) \;=\; f[h_1(v_1, \cdots, v_n), \bullet\bullet\bullet, h_k(v_1, \cdots, v_n)] \;,$$

with domain $D' = \{ \vec{v} \in D_1 \cap \cdots \cap D_k \mid [h_1(\vec{v}), \bullet\bullet\bullet, h_k(\vec{v})] \in D \}$;
and it must be proved that, for all $\vec{v} \in D'$ we have

$$\mathcal{N} \vdash A_1 \leftrightarrow A_7 \;\;,$$

in the following formula list, where, in A_5 to A_7, the string $\exists y_*$ is short for $\exists y_1 \exists y_2 \cdots \exists y_k$. Also, we suppress associativity brackets for $\wedge$, using many iterations of $\vdash F_1 \wedge (F_2 \wedge F_3) \leftrightarrow (F_1 \wedge F_2) \wedge F_3$.

$\underline{A_1 \text{ is :}}$ $\quad x_0 \approx \text{“}g(\vec{v})\text{”}$.

$\underline{A_2 \text{ is :}}$ $\quad x_0 \approx \text{“}f(w_*)\text{”}$.

$\underline{A_3 \text{ is :}}$ $\quad F^{[x_{*\overrightarrow{}} \to \text{“}w_*\text{”}]}$.

$\underline{A_4 \text{ is :}}$ $\quad G^{[y_{*\overrightarrow{}} \to \text{“}w_*\text{”}]}$.

$\underline{A_5 \text{ is :}}$ $\quad \exists y_*(y_1 \approx \text{“}w_1\text{”} \wedge \cdots\cdots \wedge y_k \approx \text{“}w_k\text{”} \wedge G)$

$\underline{A_6 \text{ is :}}$ $\quad \exists y_*(G_1^{[\vec{x} \to \text{“ } \vec{v} \text{ ”}]} \wedge \cdots \wedge G_k^{[\vec{x} \to \text{“ } \vec{v} \text{ ”}]} \wedge G)$

$\underline{A_7 \text{ is :}}$ $\quad (\exists y_*(G_1 \wedge \cdots \wedge G_k \wedge G))^{[\vec{x} \to \text{“ } \vec{v} \text{ ”}]}$.

By hypothetical syllogism applied $2 \cdot 6 = 12$ times, it is sufficient to prove that, for each $\vec{v} \in D'$, there is a $w_* \in D$ such that

$$\mathcal{N} \vdash A_i \leftrightarrow A_{i+1} \qquad \text{for } 7 > i > 0_{\mathbf{N}} \quad .$$

Of course we take $w_* = (h_1(\vec{v}), \cdots, h_k(\vec{v}))$ for this.

$\underline{\emptyset \ \vdash \ A_1 \leftrightarrow A_2}$ is clear because A_1 and A_2 are the same formula.

$\underline{\mathcal{N} \ \vdash \ A_2 \leftrightarrow A_3}$ is immediate because F represents f. See $(**)$.

$\underline{\emptyset \ \vdash \ A_3 \leftrightarrow A_4}$ is again because they are the same formula.

$\underline{\emptyset \ \vdash \ A_4 \leftrightarrow A_5}$ is really a general fact. When $k = 1$, we use $\vdash \quad J^{[x \to t]} \leftrightarrow \exists x(x \approx t \wedge J)$ (for t substitutable for x in J). This fact is easily iterated to the general case. For example (taking t_i as fixed (variable-free) terms, for simplicity),

$$\vdash \quad J^{[x_1 \to t_1][x_2 \to t_2]} \leftrightarrow \exists x_1 \exists x_2 (x_1 \approx t_1 \wedge x_2 \approx t_2 \wedge J) \ ,$$

$\underline{\mathcal{N} \ \vdash \ A_5 \leftrightarrow A_6}$ is combining all the derivations in $(*)_i$, viz.

$$\mathcal{N} \ \vdash \ (x_0 \approx \text{“}w_i\text{”} \leftrightarrow H_i^{[\vec{x} \to \text{“} \ \vec{v} \ \text{”}]})^{[x_0 \to y_i]} \ ,$$

It uses some more hypothetical syllogism and the general facts that

$$\Gamma \vdash J \to K \ \Rightarrow \ \Gamma \vdash J \wedge L \to K \wedge L \qquad \text{and} \qquad \Gamma \vdash J \ \Rightarrow \ \Gamma \vdash \exists x J \ .$$

$\underline{\emptyset \ \vdash \ A_6 \leftrightarrow A_7}$ is again because they are the same formula—note that, since $x_1, \cdots, x_n$ don't occur freely in G, that formula coincides with $G^{[\vec{x} \to \text{“} \ \vec{v} \ \text{”}]}$.

This completes the proof that a composition of representable functions is representable.

Minimization:

Suppose we are given that F represents the function $f : D \to \mathbf{N}$, where $D \subset \mathbf{N}^k$ with $k > 1$. Thus, for all $\vec{v} \in D$, we have

$$\mathcal{N} \ \vdash \ (x_0 \approx \text{“}f(\vec{v})\text{”} \leftrightarrow F^{[\vec{x} \to \text{“} \ \vec{v} \ \text{”}]}) \ .$$

Let

$$g(v_1, \cdots, v_{k-1}) \ := \ \min\{v \ : \ f(v_1, \cdots, v_{k-1}, v) = 0_{\mathbf{N}} \} \ ,$$

where the domain of g is

$$D' \ = \ \{ \, v_* \in \mathbf{N}^{k-1} \ : \ \exists v \in \mathbf{N} [(\forall w \in \mathbf{N} \, , \ v \geq w \Rightarrow (v_*, w) \in D] \text{ and } f(v_*, v) = 0_{\mathbf{N}}] \, \} \ .$$

We shall prove that g is represented by G, where

$$G \;:=\; F^{[x_{\overline{0}}\to 0][x_{\overline{k}}\to x_0]} \;\wedge\; \forall y(y < x_0 \to \neg F^{[x_{\overline{0}}\to 0\ ,\ x_{\overline{k}}\to y]}) \quad ,$$

in which y can be any variable not occurring in F. So we must show that

$$\mathcal{N} \;\vdash\; (x_0 \approx \text{“}g(v_*)\text{”} \leftrightarrow G^{[x_{\overline{*}}\to \text{“}v_*\text{”}]}) \quad .$$

The following divides the work into pieces.

Lemma V-1.1. *Going back to the general notation in the definition, a formula F represents a function f if and only if the following two conditions hold :*

$$\mathcal{N} \;\vdash\; F^{[\vec{x}\to \text{“}\ \vec{v}\ \text{”}\ ,\ x_{\overline{0}}\to \text{“}f(\vec{v})\text{”}]} \quad \text{and}$$
$$\mathcal{N} \;\vdash\; F^{[\vec{x}\to \text{“}\ \vec{v}\ \text{”}]} \to x_0 \approx \text{“}f(\vec{v})\text{”} \quad .$$

Proof. Recalling that $\leftrightarrow$ is really “ a $\wedge$ of two $\to$'s ”, and taking

$$H = F^{[\vec{x}\to \text{“}\ \vec{v}\ \text{”}]} \quad \text{and} \quad t = \text{“}f(\vec{v})\text{”}$$

below, it suffices to show, whenever t is substitutable for x in H, that

$$\Gamma \;\vdash\; H^{[x\to t]} \quad \Longleftrightarrow \qquad \Gamma \;\vdash\; (x \approx t \to H) \quad .$$

This is elementary. See Exercise 7.11 in [**LM**], and consider the contrapositive of the logical axiom $x \approx t \wedge \neg H \to \neg H^{[x\to t]}$ to prove $\Rightarrow$. For the reverse implication, just substitute t for x in the hypothesis.

By the “if” half of this lemma, it remains to prove, for all $v_* \in D'$, that

$$\text{(i)} \qquad \mathcal{N} \;\vdash\; G^{[x_{\overline{*}}\to \text{“}v_*\text{”}\ ,\ x_{\overline{0}}\to \text{“}g(v_*)\text{”}]} \quad \text{and}$$
$$\text{(ii)} \qquad \mathcal{N} \;\vdash\; G^{[x_{\overline{*}}\to \text{“}v_*\text{”}]} \to x_0 \approx \text{“}g(v_*)\text{”} \quad .$$

By the “only if” half, applied to the fact that F represents f, we get

$$\mathcal{N} \;\vdash\; F^{[\vec{x}\to \text{“}\ \vec{v}\ \text{”}\ ,\ x_{\overline{0}}\to \text{“}f(\vec{v})\text{”}]}$$

for all $\vec{v} \in D$. So taking $\vec{v} = (v_*, g(v_*))$ which yields $f(\vec{v}) = 0_{\mathbf{N}}$, the above display becomes

$$\mathcal{N} \;\vdash\; F^{[x_{\overline{0}}\to 0\ ,\ x_{\overline{*}}\to \text{“}v_*\text{”}\ ,\ x_{\overline{k}}\to \text{“}g(v_*)\text{”}]} \qquad (*)$$

This does half of (i), since its formula is a $\wedge$ of two formulae; and the other half of (i) reduces to

$$\text{(iii)} \qquad \mathcal{N} \vdash \forall y(y < \text{“}g(v_*)\text{”} \to \neg F^{[x_{\vec{0}} \to 0\ ,\ x_{\vec{*}} \to \text{“}v_*\text{”}\ ,\ x_{\vec{k}} \to y]}) \ .$$

So it's now (ii) and (iii) that remain to be proved. For (ii), use the following propositional fact :

If $\Gamma \vdash A \vee B \vee C$, $\Gamma \vdash D \to \neg A$ *and* $\Gamma \vdash E \to \neg C$, *then* $\Gamma \vdash D \wedge E \to B$.

To see this, loosely speaking,

$$\Gamma \cup \{D \wedge E\} \vdash \{A \vee B \vee C\ ,\ \neg A\ ,\ \neg C\} \vdash B \ ,$$

and now use the deduction lemma. This propositional fact, plus (iv), (v) and (vi) below, give (ii) immediately.

$$\text{(iv)} \qquad \mathcal{N} \vdash (x_0 < \text{“}g(v_*)\text{”} \ \vee \ x_0 \approx \text{“}g(v_*)\text{”} \ \vee \ \text{“}g(v_*)\text{”} < x_0) \ .$$

$$\text{(v)} \qquad \mathcal{N} \vdash F^{[x_{\vec{0}} \to 0][x_{\vec{*}} \to \text{“}v_*\text{”}\ ,\ x_{\vec{k}} \to x_0]} \ \to \ \neg x_0 < \text{“}g(v_*)\text{”}.$$

$$\text{(vi)} \quad \mathcal{N} \vdash \forall y(y < x_0 \to \neg F^{[x_{\vec{0}} \to 0\ ,\ x_{\vec{*}} \to \text{“}v_*\text{”}\ ,\ x_{\vec{k}} \to y]}) \ \to \ \neg \text{“}g(v_*)\text{”} < x_0 \ .$$

So we now have <u>four</u> things, (iii)–(vi), to prove, instead of two, but there is light at the end of the tunnel !

Claim (iv) is an example of (IV) from the list of $\mathcal{N}$-specific derivations.

For (v) we use (VII) in the previous list of $\mathcal{N}$-specific derivations, which allows us to get a derivation from $\mathcal{N}$ of $(\neg J_0 \to \neg x < \text{“}\ell\text{”})$, if we have such derivations of $J_0^{[x \to \text{“}i\text{”}]}$ for all i in the range $\ell > i \geq 0$. Thus, taking $J_0 = F^{[x_{\vec{0}} \to 0][\vec{x} \to (\text{“}v_*\text{”},\ x_0)]}$ and $x = x_0$, we need derivations of

$$\neg F^{[x_{\vec{0}} \to 0][\vec{x} \to (\text{“}v_*\text{”},\ x_0)][x_{\vec{0}} \to \text{“}i\text{”}]} \ , \quad \text{which equals} \quad \neg F^{[\vec{x} \to (\text{“}v_*\text{”}, \text{“}i\text{”})][x_{\vec{0}} \to 0]} \qquad (**)$$

for all i in the range $g(v_*) > i \geq 0$. But taking $\vec{v} = (v_*, i)$ in half the basic assumption about F at the beginning, and applying substitution of 0 for x_0, [recalling that $\Gamma \vdash (K \to L)$ implies that $\Gamma \vdash (K^{[x \to t]} \to L^{[x \to t]})$], we obtain

$$\mathcal{N} \vdash \neg x_0 \approx \text{“}f(v_*, i)\text{”}^{[x_{\vec{0}} \to 0]} \ \to \ \neg F^{[\vec{x} \to (\text{“}v_*\text{”}, \text{“}i\text{”})][x_{\vec{0}} \to 0]} \ .$$

By modus ponens, we need only a derivation from $\mathcal{N}$ of the formula just above to the left of the arrow, and that formula is simply $\neg\ 0 \approx \text{“}f(v_*, i)\text{”}$.

This is a case of (III) in the previous list of $\mathcal{N}$-specific derivations, in view of the fact that $f(v_*, i) \neq 0_{\mathbf{N}}$ for all the values of i at issue.

In (vi), the formula we need to derive is

$$\forall y(y < x_0 \to \neg J) \quad \to \quad \neg \text{ “}g(v_*)\text{”} < x_0 \quad ,$$

where $J = F^{[\vec{x} \to (\text{“}v_*\text{”},\ y)\ ,\ x_{\vec{0}} \to 0]}$. This is 'propositionally' immediately transformed, using the definition of $\exists$, to requiring a derivation from $\mathcal{N}$ of the formula

$$\exists y(y < x_0 \wedge J) \quad \vee \quad \neg \text{ “}g(v_*)\text{”} < x_0 \quad .$$

The quantifier can range over the whole formula, and then general properties of quantifiers give that it suffices to find some fixed term t for which there is a derivation from $\mathcal{N}$ of the formula

$$((y < x_0 \wedge J) \quad \vee \quad \neg \text{ “}g(v_*)\text{”} < x_0)^{[y \to t]} \quad .$$

Taking t to be “$g(v_*)$”, and since $J \ \to \ (K \wedge J) \vee \neg K$ is a tautology, (MP) reduces it to finding a derivation from $\mathcal{N}$ of the formula $J^{[y \to t]}$. But the latter formula is $F^{[\vec{x} \to (\text{“}v_*\text{”},\ \text{“}g(v_*)\text{”})\ ,\ x_{\vec{0}} \to 0]}$. We already noted the existence of this derivation in proving half of (i) [see $(*)$].

Finally to prove (iii), we can drop the universal quantifier by the rule of inference (GEN), and (using J from the previous paragraph) this asks for a derivation from $\mathcal{N}$ of the formula $y < \text{“}g(v_*)\text{”} \to \neg J$. Again using (VII) in the previous list of $\mathcal{N}$-specific derivations, it suffices to find derivations of $\neg J^{[y \to \text{“}i\text{”}]}$ for all i in the range $g(v_*) > i \geq 0$. But that's exactly what we did in the proof of (v) [see $(**)$].

This completes the proof that the minimization of a representable function is representable.

Here is the result connecting representability and expressibility.

Proposition V-1.2. *Let $h : D \to \mathbf{N}$ be any 01-function. Then h is representable if and only if it is expressible.*

Proof. First suppose that F represents h. Let $G \ := \ F^{[x_{\vec{0}} \to 0]}$. To show that G expresses h, let $\vec{v} \in D$.

In the case $h(\vec{v}) = 0_{\mathbf{N}}$, we have $\mathcal{N} \vdash x_0 \approx 0 \leftrightarrow F^{[\vec{x}\rightarrow\text{“ }\vec{v}\text{ ”}]}$. Using the $\rightarrow$ half, and since, at least for fixed terms t we have

$$\Gamma \vdash J \rightarrow K \quad \Rightarrow \quad \Gamma \vdash J^{[x\rightarrow t]} \rightarrow K^{[x\rightarrow t]} \ ,$$

we get

$$\mathcal{N} \vdash (x_0 \approx 0)^{[x_{\overline{0}}\rightarrow 0]} \rightarrow F^{[\vec{x}\rightarrow\text{“ }\vec{v}\text{ ”}][x_{\overline{0}}\rightarrow 0]} \ .$$

The latter formula is $0 \approx 0 \rightarrow G^{[\vec{x}\rightarrow\text{“ }\vec{v}\text{ ”}]}$. By (MP) and since $\vdash 0 \approx 0$, we get $\mathcal{N} \vdash G^{[\vec{x}\rightarrow\text{“ }\vec{v}\text{ ”}]}$, as required.

In the case $h(\vec{v}) = 1_{\mathbf{N}}$, we have $\mathcal{N} \vdash x_0 \approx 1 \leftrightarrow F^{[\vec{x}\rightarrow\text{“ }\vec{v}\text{ ”}]}$. Thus, $\mathcal{N} \vdash \neg x_0 \approx 1 \rightarrow \neg F^{[\vec{x}\rightarrow\text{“ }\vec{v}\text{ ”}]}$ (contrapositive of the $\leftarrow$ half). And so,

$$\mathcal{N} \vdash (\neg x_0 \approx 1)^{[x_{\overline{0}}\rightarrow 0]} \rightarrow \neg\, F^{[\vec{x}\rightarrow\text{“ }\vec{v}\text{ ”}][x_{\overline{0}}\rightarrow 0]} \ ,$$

that is, $\mathcal{N} \vdash \neg 0 \approx 1 \rightarrow \neg G^{[\vec{x}\rightarrow\text{“ }\vec{v}\text{ ”}]}$.

But $\neg 0 \approx 1$ is the same formula as $\neg$“$0_{\mathbf{N}}$” $\approx$ “$1_{\mathbf{N}}$” . And $0_{\mathbf{N}} \neq 1_{\mathbf{N}}$, according to the latest calculations ! So using (III) in the previous list of derivations from $\mathcal{N}$, we have $\mathcal{N} \vdash \neg 0 \approx 1$. By (MP), $\mathcal{N} \vdash \neg G^{[\vec{x}\rightarrow\text{“ }\vec{v}\text{ ”}]}$, as required.

Conversely suppose that G expresses h. Define

$$F \ := \ (G \ \wedge \ x_0 \approx 0) \ \vee \ (\neg G \ \wedge \ x_0 \approx 1) \ .$$

To show that F represents h , let $\vec{v} \in D$.

In the case $h(\vec{v}) = 0_{\mathbf{N}}$, we have $\mathcal{N} \vdash G^{[\vec{x}\rightarrow\text{“ }\vec{v}\text{ ”}]}$. Now

$$F^{[\vec{x}\rightarrow\text{“ }\vec{v}\text{ ”}]} \ = \ (G^{[\vec{x}\rightarrow\text{“ }\vec{v}\text{ ”}]} \wedge x_0 \approx 0) \ \vee \ (\neg\, G^{[\vec{x}\rightarrow\text{“ }\vec{v}\text{ ”}]} \wedge x_0 \approx 1) \ .$$

The following are easy general ‘propositional’ facts:

$$\Gamma \vdash F \rightarrow H_1 \ \Rightarrow \ \Gamma \vdash F \rightarrow H_1 \vee H_2 \quad ; \quad \Gamma \vdash J \ \Rightarrow \ \Gamma \vdash F \rightarrow J \ ;$$

$$\Gamma \vdash F \rightarrow F \quad ; \quad [\Gamma \vdash F \rightarrow J \ \text{ and } \ \Gamma \vdash F \rightarrow K] \ \Rightarrow \ \Gamma \vdash F \rightarrow K \wedge J \ .$$

Taking F and K to be $x_0 \approx 0$, taking J to be $G^{[\vec{x}\rightarrow\text{“ }\vec{v}\text{ ”}]}$, taking H_1 to be $J \wedge x_0 \approx 0$, and taking H_2 to be $\neg J \wedge x_0 \approx 1$, this does the required job of showing the more difficult half (namely, the $\rightarrow$) of

$$\mathcal{N} \vdash x_0 \approx 0 \leftrightarrow F^{[\vec{x}\rightarrow\text{“ }\vec{v}\text{ ”}]} \ .$$

In the case $h(\vec{v}) = 1_{\mathbf{N}}$, we have $\mathcal{N} \ \vdash \ \neg G^{[\vec{x} \to \text{“ } \vec{v} \text{ ”}]}$. The argument then proceeds almost exactly as in the previous paragraph.

Combining this proposition with the main argument preceding it, we get, modulo the following appendix, the following central result.

Theorem V-1.3 (Gödel). *Any recursive function is representable. Any recursive* 01*-function, and, in particular, any recursive relation, is expressible.*

APPENDIX : Derivations from $\mathcal{N}$.

We shall prove (I) to (VII) from near the beginning of this section. First here is the list $\mathcal{N}$:

$\mathcal{N}1.$ $\quad \forall x \ \neg Sx \approx 0$
$\mathcal{N}2.$ $\quad \forall x \forall y \ (Sx \approx Sy \ \rightarrow \ x \approx y)$
$\mathcal{N}3.$ $\quad \forall x \ x + 0 \approx x$
$\mathcal{N}4.$ $\quad \forall x \forall y \ S(x + y) \approx x + Sy$
$\mathcal{N}5.$ $\quad \forall x \ x \times 0 \approx 0$
$\mathcal{N}6.$ $\quad \forall x \forall y \ x \times Sy \approx (x \times y) + x$
$\mathcal{N}7.$ $\quad \forall x \ \neg x < 0$
$\mathcal{N}8.$ $\quad \forall x \forall y \ (x < Sy \ \rightarrow \ (x < y \ \vee \ x \approx y))$
$\mathcal{N}9.$ $\quad \forall x \forall y \ (x < y \ \vee \ x \approx y \ \vee \ y < x)$

We take $x = x_1$ and $y = x_2$, so that these become nine completely specific sentences in the language of $1^{\underline{\text{st}}}$order number theory. But then it is almost immediate from the workhorse logical axiom $\forall x F \ \rightarrow F$ and (MP) that all these formulae with the quantifiers removed lie in $\mathcal{N}^{\vdash}$; and from the workhorse axiom $\forall x F \ \rightarrow F^{[x \to z]}$, for variables z not occurring in F, that the same holds with x and y being *any* two distinct variables. But now the rule of inference (GEN) allows us to 'assume' $\mathcal{N}1$ to $\mathcal{N}9$ (*with* the quantifiers) for any two distinct variables x and y .

Proof of (I) : Proceed by induction on b to show

$$\mathcal{N} \ \vdash \ \text{“}a\text{”} + \text{“}b\text{”} \ \approx \ \text{“}a +_{\mathbf{N}} b\text{”}$$

for all a and b.

When $b = 0_{\mathbf{N}}$, we need a derivation of "a" + "$0_{\mathbf{N}}$" $\approx$ "$a +_{\mathbf{N}} 0_{\mathbf{N}}$" , that is, of "$a$" $+ 0 \approx$ "a". This is immediate from $\mathcal{N}3$ using the workhorse logical axiom and (MP) as just above.

For the inductive step, we are given a derivation for
"a" + "b" $\approx$ "$a +_{\mathbf{N}} b$", and we need to derive
"a"+"$b+_{\mathbf{N}}1_{\mathbf{N}}$" $\approx$ "$a+_{\mathbf{N}}(b+_{\mathbf{N}}1_{\mathbf{N}})$" , that is, "$a$"+$S$"$b$" $\approx$ S"$a+_{\mathbf{N}}b$". This is immediate from $\mathcal{N}4$ and the inductive hypothesis, by the same argument with the workhorse axiom and (MP) (and also using the logical axioms that may be sloganized as : *'$\approx$ is symmetric'* and *'$\approx$ is transitive'* and *'replacements of equals in terms yields equals'*).

Proof of (II) : Again proceed by induction on b to show

$$\mathcal{N} \ \vdash \ \text{"}a\text{"} \times \text{"}b\text{"} \ \approx \ \text{"}ab\text{"}$$

for all a and b.

When $b = 0_{\mathbf{N}}$, we need a derivation of
"a" $\times$ "$0_{\mathbf{N}}$" $\approx$ "$a0_{\mathbf{N}}$" , that is, of "a" $\times 0 \approx 0$. This is immediate from $\mathcal{N}5$, using the workhorse axiom and (MP) as above.

For the inductive step, we are given a derivation for "a" $\times$ "b" $\approx$ "ab" and we need to derive

"a" $\times$ "$b +_{\mathbf{N}} 1_{\mathbf{N}}$" $\approx$ "$a(b +_{\mathbf{N}} 1_{\mathbf{N}})$" , that is, "$a$" $\times$ S"b" $\approx$ "$ab +_{\mathbf{N}} a$".

By $\mathcal{N}6$, we have a derivation of "a" $\times$ S"b" $\approx$ ("a" $\times$ "b") + "a".

Using the inductive hypothesis, we get a derivation of

("a" $\times$ "b") + "a" $\approx$ "ab" + "a" .

This also uses the logical axiom which may be sloganized as *'replacements of equals in terms yields equals'*.

By (I) we have a derivation of "ab" + "a" $\approx$ "$ab +_{\mathbf{N}} a$" .

Since *transitivity of* $\approx$ is part of the logical axioms, combining the last three, we're done.

Proof of (III) : Proceed by induction on a to prove that

$$\mathcal{N} \ \vdash \ \neg \text{"}a\text{"} \ \approx \ \text{"}b\text{"} \qquad \text{for any} \ \ a > b \ \text{in} \ \mathbf{N} \ .$$

This suffices, by the *symmetry of* $\approx$ *axiom*, to prove it for all $a \neq b$.

The initial case is vacuous.

Inductively assume that, for $a > c$, we can derive $\neg$"a" $\approx$ "c" .

Now assume $a +_{\mathbf{N}} 1_{\mathbf{N}} > b$. We must deduce, since "$a +_{\mathbf{N}} 1_{\mathbf{N}}$" is S"a", that there is a derivation of $\neg\ S$"a" $\approx$ "b" .

For $b = 0_{\mathbf{N}}$, we get this directly from $\mathcal{N}1$.

Otherwise, write $b = c +_{\mathbf{N}} 1_{\mathbf{N}}$. So we need derivation of $\neg\ S$"a" $\approx S$"c". Since $a > c$, induction gives a derivation of $\neg$ "a" $\approx$ "c" . By $\mathcal{N}2$ and the contapositive, we have a derivation of $\neg$ "a" $\approx$ "c" $\ \rightarrow\ \neg\ S$"a" $\approx S$"c" . Thus an application of (MP) does the job.

Proof of (IV): This result, namely that

$$\mathcal{N} \quad \vdash \quad t < s \ \vee\ t \approx s \ \vee\ s < t \qquad \text{for any terms } \ s \text{ and } t \ ,$$

is almost immediate from $\mathcal{N}9$, using the 'workhorse' logical axiom and (MP).

Proof of (VI): By induction on b, we shall show, for all $a \geq b$, that $\mathcal{N} \vdash \neg$ "a" $<$ "b" .

When $b = 0_{\mathbf{N}}$, the formula $\neg$ "a" < 0 is derivable immediately from $\mathcal{N}7$ in the usual way.

Inductively assuming the statement, now take numbers $a \geq b +_{\mathbf{N}} 1_{\mathbf{N}}$, and show $\mathcal{N} \vdash \neg$ "a" $< S$"b", since "$b +_{\mathbf{N}} 1_{\mathbf{N}}$" is just S"b". By $\mathcal{N}8$ and the contrapositive, we have

$$\mathcal{N} \ \vdash\ \neg(\text{"}a\text{"} < \text{"}b\text{"} \vee \text{"}a\text{"} \approx \text{"}b\text{"}) \rightarrow \neg\ \text{"}a\text{"} < S\text{"}b\text{"} \ .$$

It remains to find a derivation of the formula to the left of the arrow in the display. Now, 'propositionally' quite generally we have

$$[\ \Gamma \vdash \neg F \text{ and } \Gamma \vdash \neg G\ ] \ \ \Rightarrow \ \ \Gamma \vdash \neg F \wedge \neg G \ \ \Rightarrow \ \ \Gamma \vdash \neg(F \vee G) \ .$$

This leaves us to find two derivations showing :

(i) $\mathcal{N} \ \vdash \ \neg$"$a$" $<$ "b" , which is immediate by the inductive hypothesis, since $a \geq b +_{\mathbf{N}} 1_{\mathbf{N}}$ certainly gives $a \geq b$; and

(ii) $\mathcal{N} \ \vdash \ \neg$"$a$" $\approx$ "b" , which is immediate from (III), since $a \geq b +_{\mathbf{N}} 1_{\mathbf{N}}$ certainly gives $a \neq b$.

Proof of (V): For $b > a$, we need to show $\mathcal{N} \ \vdash$ "a" $<$ "b" . Now

$$\mathcal{N} \ \vdash \ \text{"}a\text{"} < \text{"}b\text{"} \ \vee \ \text{"}a\text{"} \approx \text{"}b\text{"} \ \vee \ \text{"}b\text{"} < \text{"}a\text{"}$$

by $\mathcal{N}9$ or (IV). And generally 'propositionally',

$$[\ \Gamma \vdash F \vee G \vee H \text{ and } \Gamma \vdash \neg G \text{ and } \Gamma \vdash \neg H\ ] \ \Rightarrow\ \Gamma \vdash F\ .$$

This leaves us to find two derivations showing :

(i) $\mathcal{N}\ \vdash\ \neg\text{“}a\text{”} \approx \text{“}b\text{”}$, which is immediate from (III), since $a \neq b$; and

(ii) $\mathcal{N}\ \vdash\ \neg\text{“}b\text{”} < \text{“}a\text{”}$, which is immediate from (VI), since $b \geq a$.

Proof of (VII): This states that, if $\mathcal{N} \vdash J^{[x\to\text{“}i\text{”}]}$ for all i with $\ell > i \geq 0$, then $\mathcal{N}\ \vdash\ x < \text{“}\ell\text{”}\ \rightarrow\ J$. We proceed by induction on ℓ.

When $\ell = 0_{\mathbf{N}}$, use the propositional facts that

$$\Gamma \vdash \neg K\ \ \Rightarrow\ \ \Gamma \vdash \neg K \vee J\ \ \Rightarrow\ \ \Gamma \vdash K \rightarrow J\ ,$$

leaving us to prove $\mathcal{N} \vdash \neg\ x < 0$, immediate from $\mathcal{N}7$.

For the inductive step, assume the statement and that $\mathcal{N} \vdash J^{[x\to\text{“}i\text{”}]}$ for all i with $\ell +_{\mathbf{N}} 1_{\mathbf{N}} > i \geq 0$. We need to show $\mathcal{N}\ \vdash\ x < \text{“}\ell +_{\mathbf{N}} 1_{\mathbf{N}}\text{”}\ \rightarrow\ J$. Since $\text{“}\ell +_{\mathbf{N}} 1_{\mathbf{N}}\text{”}$ is just $S\text{“}\ell\text{”}$, and since by $\mathcal{N}8$

$$\mathcal{N}\ \vdash\ x < S\text{“}\ell\text{”}\ \rightarrow x < \text{“}\ell\text{”} \vee x \approx \text{“}\ell\text{”}\ ,$$

hypothetical syllogism leaves us to prove $\mathcal{N}\ \vdash\ x < \text{“}\ell\text{”} \vee x \approx \text{“}\ell\text{”}\ \rightarrow\ J$. A 'propositional' fact is that

$$[\ \Gamma \vdash F \rightarrow J\ \text{ and }\ \Gamma \vdash G \rightarrow J\ ]\ \Rightarrow\ \Gamma \vdash F \vee G\ \rightarrow\ J\ .$$

By the inductive assumption, $\mathcal{N}\ \vdash\ x < \text{“}\ell\text{”}\ \rightarrow\ J$. So we are left to show $\mathcal{N}\ \vdash\ x \approx \text{“}\ell\text{”}\ \rightarrow\ J$. But, quite generally,

$$\Gamma\ \vdash\ H^{[x\to t]}\ \ \Longrightarrow\ \ \ \ \Gamma\ \vdash\ (x \approx t \rightarrow H)\ .$$

(See half the proof of the Lemma **V-1.1**.) We were given $\mathcal{N} \vdash J^{[x\to\text{“}i\text{”}]}$ for $\ell \geq i$, and in particular for $\ell = i$, so this completes the proof.

V-2: Semi-decidability and semi-expressibility.

This subsection and the next are somewhat peripheral to the main purpose here, so will be relegated to small print. However, a few points from [**LM**], unsubstantiated there, will be discussed here. The main one relates to technicalities connected to the 'Berry paradox proof' by Boolos of Gödel's incompleteness theorem.

We shall deal with relations and, more generally, with 01-functions. We shall also drop the subscripts on $0_{\mathbf{N}}$, etc. from now on, and trust the reader to keep in mind the distinction between logical symbols and numbers.

Given a subset S of $\mathbf{N}^n$ for some n, the following notation for two closely related functions (a relation f, and a partial function g) will be used throughout this subsection. Define

$$g \ : \ \vec{w} \ \mapsto \ \begin{cases} 1 & \text{if } \vec{w} \in S \ ; \\ \text{undefined} & \text{if } \vec{w} \notin S \ ; \end{cases}$$

and

$$f \ : \ \vec{w} \ \mapsto \ \begin{cases} 1 & \text{if } \vec{w} \in S \ ; \\ 0 & \text{if } \vec{w} \notin S \ . \end{cases}$$

We shall say that the relation f is *decidable* iff f is computable (i.e. recursive), and that f is *semi-decidable* iff g is computable.

An excellent (and historically crucial) example to keep in mind is taking f to be the halting function, which Turing's fundamental result showed to be non-computable. See Theorem **IV-1.5**, where the corresponding function g was noted to be computable just before the statement of that theorem. Thus we do have semi-decidable relations which are not decidable, another version of that most fundamental fact discovered about computability. Note that decidability implies semi-decidability.

In a notation extended from Subsection **IV-5**, the relation f is decidable exactly when the set S is recursive, and f is semi-decidable exactly when S is recursively enumerable. In that subsection we only applied these adjectives to subsets of $\mathbf{N}$. For subsets of $\mathbf{N}^n$, the definition of S being recursive is the same, just that f above is a recursive function. The most convenient definition of S being recursively enumerable is that it is the domain of at least one recursive function. An easy exercise shows that S is recursively enumerable if and only if $c(S)$ is recursively enumerable for some recursive bijection $c : \mathbf{N}^n \to \mathbf{N}$. We'll apply this below with $c = CAN$, Cantor's function from Appendix A.

In the example above, the notation used has our $\vec{w}$ as $(c, \vec{v})$. Furthermore, for that example, there is a relation h of one more variable (namely, Kleene's relation KLN) such that $f(\vec{w})$ holds if and only if there exists some number k such that $h(\vec{w}, k)$ holds. (The number k would be the code of the history of the computation. So note that h has changed its meaning from its use in the Section **IV** example.) We shall see below that, quite generally, the semi-decidable relations are exactly the ones for which a decidable relation h as above exists.

First let's do an informal version. Given a relation h in "$n + 1$" variables, and an algorithm to compute it, *define* f, a relation in "n" variables, by

$f(\vec{w})$ if and only if there exists some number k such that $h(\vec{w}, k)$.

To see informally that f is semi-decidable, we need an algorithm to compute the corresponding g in the notation above. For this, just run the algorithm for $h(\vec{w}, 0)$, then for $h(\vec{w}, 1)$, then for $h(\vec{w}, 2)$, etc., stopping only when and if one gets the answer 1. Then 'output' that 1. Of course, we get an 'infinite loop' exactly where we want one.

Conversely, suppose given a semi-decidable f, so that one has an algorithm for the corresponding g, or, more to the point here, the set S in the general notation from the beginning is recursively enumerable. Let some algorithmic listing of S be fixed, and now define a relation h in one extra variable by

$$h(\vec{w}, k) = 1 \iff \vec{w} \text{ is the } k\text{th member of the list}$$

(where the list starts with its 0th member!)

Then clearly we have

$$f(\vec{w}) = 1 \iff \exists k \text{ with } h(\vec{w}, k) = 1 \ ,$$

so that h and f stand in the correct relation to each other. Furthermore (continuing in the informal sense), the relation h is certainly decidable, since, given $(\vec{w}, k)$, one computes h on that input by just running the listing algorithm until it has produced its kth piece of output, and by then comparing the latter to $\vec{w}$, 'outputting' 1 if they agree, and 0 if they don't.

These arguments, and several below in this subsection, fill in details for a number of results stated without proof on pages 402-403 at the end of [**LM**]. This is yet another instance where it is hardly any more difficult to gave exact details from the **BTEN** characterization of computability (rather than caving in completely and just appealing to Church's thesis and the arguments above). We'll write down some **BTEN** commands to prove the theorem below, and partly rely on the above informal descriptions to make it clear that these algorithms do what they are supposed to do. Also, the point above about a subset of $\mathbf{N}^n$ being algorithmically listable is made more convincing (when $n > 1$) by actually writing out the commands as we do below.

Theorem V-2.1. (i) *If a relation f in "n" variables is semi-decidable, then there is a decidable relation h in "$n+1$" variables such that*

$$f(\vec{w}) = 1 \iff \exists k \ h(\vec{w}, k) = 1 \ .$$

(ii) *Conversely, given a decidable relation h in "$n+1$" variables, the relation f defined by the above display is semi-decidable.*

Proof. (ii) A command for computing the partial function g corresponding to f is the following, where $C[h]$ is one for strongly computing h :

($x_{n+1} \leftarrow 0$; $\mathcal{B}_{1,\cdots,n+1} C[h] \mathcal{E}$;
while $\neg x_0 \approx 1$
do ($x_{n+1} \leftarrow x_{n+1} + 1$; $\mathcal{B}_{1,\cdots,n+1} C[h] \mathcal{E}$))

(i) Using the fact that $f^{-1}(1)$ is a recursively enumerable set, and applying Theorem **IV-5.1**, let ℓ (for 'list') be a computable total function which 'lists' $CAN_n(f^{-1}(1))$; that is, the latter subset of $\mathbf{N}$ is the image of ℓ . Recall from Appendix A that CAN_n is the recursive bijection of Cantor from $\mathbf{N}^n$ to $\mathbf{N}$. Let $C[\ell]$ be a command which strongly computes ℓ .

Consider the following command (the column on the left) :

$\underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}(0, w_1, \cdots, w_n, k, - - -)$

$(\mathcal{B}_{n+1}C[CAN_n]\mathcal{E}$;

$\underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}(CAN_n(w_1, \cdots, w_n), -, \cdots, -, k, - - -)$

$x_{n+2} \Leftarrow x_0$;

$\underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}(-, \cdots, -, k, CAN_n(w_1, \cdots, w_n), - - - -)$

$x_1 \Leftarrow x_{n+1}$;

$\underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}(-, k, -, \cdots, -, CAN_n(w_1, \cdots, w_n), - - -)$

$\mathcal{B}_{n+2}C[\ell]\mathcal{E}$;

$\underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}(\ell(k), -, \cdots, -, CAN_n(w_1, \cdots, w_n), - - -)$

if $x_0 \approx x_{n+2}$

$\underline{\qquad\qquad}(\ell(k), -, \cdots, -, \ell(k), - - -)$

thendo $x_0 \Leftarrow 1$

$\underline{\qquad\qquad}(\underline{1}, - - -) \qquad\qquad\qquad (\ell(k), -, \cdots, -, \neq \ell(k), - - -)$

elsedo $x_0 \Leftarrow 0)$

$\underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}(\underline{0}, - - -)$

The function h of "$n+1$"" variables which that command computes is the following:

$$(\vec{w}, k) \ \mapsto \ \begin{cases} 1 & \text{if } \vec{w} \text{ is } k\text{th in } \ell'\text{s list ;} \\ 0 & \text{otherwise .} \end{cases}$$

We have given a 'semantic program verification' for this on the right-hand side above, with the output double-underlined. Thus the function h is clearly a relation which has the property given in the theorem's display line, completing the proof.

What about the question of expressibility for a semi-decidable relation f ?

Since the corresponding 01-function g is computable, from the basic results in the previous subsection, g is expressible in the sense that we do get the existence of a formula G such that

$$\vec{w} \in S \implies \mathcal{N} \vdash G^{[\vec{x} \to \text{“ } \vec{w} \text{ ”}]} .$$

But a stronger result, making the implication go both ways, is obtained by considering the relation h in one extra variable studied just above.

Definition. Say that a relation f is semi-expressible (or weakly expressible) if and only if we have the existence of a formula G from 1$\underline{\text{st}}$order number theory such that

$$\vec{w} \in S \iff \mathcal{N} \vdash G^{[\vec{x} \to \text{“ } \vec{w} \text{ ”}]} .$$

i.e.

$$f(\vec{w}) = 1 \iff \mathcal{N} \vdash G^{[\vec{x} \to \text{“ } \vec{w} \text{ ”}]} .$$

Theorem V-2.2. *If a relation f is semi-decidable then it is semi-expressible, assuming that $\mathcal{N}$ is ω-consistent.* (See the proof below for the definition of ω-consistency. Note that this assumption is weaker, if anything, than assuming that all the sentences in $\mathcal{N}$ are true in $\mathbf{N}$—which seems like no assumption at all! As an easy exercise, show that expressibility implies semi-expressibility, as long as $\mathcal{N}$ is consistent.)

Proof. Using the previous theorem, let $f(\vec{w})$ be written as $\exists k\ h(\vec{w},k)$ for some decidable relation h in "$n+1$" variables. Let $\neg J$ be a formula which expresses h. Thus

$$h(\vec{w},k)=1 \implies \mathcal{N} \vdash J^{[\vec{x}\to\text{“ }\vec{w}\text{ ”}][x_{n+1}\text{“}k\text{”}]}\ , \quad \text{and}$$

$$h(\vec{w},k)=0 \implies \mathcal{N} \vdash \neg J^{[\vec{x}\to\text{“ }\vec{w}\text{ ”}][x_{n+1}\text{“}k\text{”}]}\ .$$

To show that f is semi-expressible, take $G := \exists x_{n+1}\ J$, and $S = f^{-1}(1)$, and we shall prove that

$$\vec{w}\in S \iff \mathcal{N} \vdash G^{[\vec{x}\to\text{“ }\vec{w}\text{ ”}]}\ .$$

To establish the "$\Longrightarrow$" half, we have

$$\vec{w}\in S \implies f(\vec{w})=1 \implies h(\vec{w},k)=1 \text{ for some } k \implies$$

$$\text{for some } k\ ,\ \mathcal{N} \vdash J^{[\vec{x}\to\text{“ }\vec{w}\text{ ”}]\ [x_{n+1}\text{“}k\text{”}]} \implies \mathcal{N} \vdash \exists x_{n+1}\ (J^{[\vec{x}\to\text{“ }\vec{w}\text{ ”}]})\ .$$

But the last formula is in fact $(\exists x_{n+1}\ J)^{[\vec{x}\to\text{“ }\vec{w}\text{ ”}]}) = G^{[\vec{x}\to\text{“ }\vec{w}\text{ ”}]}$, as required.

To establish the "$\Longleftarrow$" half, assume that $\mathcal{N} \vdash G^{[\vec{x}\to\text{“ }\vec{w}\text{ ”}]}$. That formula is $\neg\forall x_{n+1}\neg J^{[\vec{x}\to\text{“ }\vec{w}\text{ ”}]}$. Suppose, for a contradiction, that $\vec{w}\notin S$. Then $f(\vec{w})=0$, so $h(\vec{w},k)=0$ for all k. Therefore

$$\text{for all } k\ ,\ \mathcal{N} \vdash \neg J^{[\vec{x}\to\text{“ }\vec{w}\text{ ”}]\ [x_{n+1}\text{“}k\text{”}]}\ .$$

Taking $F=\neg J^{[\vec{x}\to\text{“ }\vec{w}\text{ ”}]}$ and $x=x_{n+1}$, we have established that $\mathcal{N} \vdash F^{[x\to\text{“}k\text{”}]}$ for all k, and also $\mathcal{N} \vdash \neg\forall x F$. This contradicts ω-consistency of $\mathcal{N}$ [whose definition says that no such pair (F,x) exists], completing the proof.

Definition. Say that a relation f is definable (or perhaps, definable in the language of $1^{\underline{\text{st}}}$order number theory) if and only if we have the existence of a formula G from $1^{\underline{\text{st}}}$order number theory such that

$$\vec{w}\in S \iff G^{[\vec{x}\to\text{“ }\vec{w}\text{ ”}]} \text{ is true in } \mathbf{N}\ .$$

$$\text{i.e.} \qquad f(\vec{w})=1 \iff G^{[\vec{x}\to\text{“ }\vec{w}\text{ ”}]} \text{ is true in } \mathbf{N}\ .$$

Theorem V-2.3. *If a relation f is semi-decidable then it is definable, assuming the obvious, that $\mathcal{N}$ is a set of true formulas in* $\mathbf{N}$ (and so is ω-consistent and consistent. Note how the proof below starts identically to the previous one.)

Proof. Using the Theorem **V-2.1**, let $f(\vec{w})$ be written as $\exists k\ h(\vec{w},k)$ for some decidable relation h in "$n+1$" variables. Let $\neg J$ be a formula which expresses h. Thus

$$h(\vec{w},k)=1 \implies \mathcal{N} \vdash J^{[\vec{x}\to\text{“ }\vec{w}\text{ ”}][x_{n+1}\text{“ }k\text{ ”}]}\ , \quad \text{and}$$

$$h(\vec{w},k)=0 \implies \mathcal{N} \vdash \neg J^{[\vec{x}\to\text{“ }\vec{w}\text{ ”}][x_{n+1}\text{“}k\text{”}]}\ .$$

To show that f is definable, take $G := \exists x_{n+1}\ J$, and $S = f^{-1}(1)$, and we shall prove that

$$\vec{w} \in S \iff G^{[\vec{x} \to \text{“ } \vec{w} \text{ ”}]} \text{ is true in } \mathbf{N} .$$

To establish the "$\Longrightarrow$" half, simply note that, as proved in the previous theorem, G semi-represents f, and that any formula which is derivable from $\mathcal{N}$ is true in $\mathbf{N}$, as required.

To establish the "$\Longleftarrow$" half, assume that $G^{[\vec{x} \to \text{“ } \vec{w} \text{ ”}]}$ is true in $\mathbf{N}$. That formula is $\neg\forall x_{n+1} \neg J^{[\vec{x} \to \text{“ } \vec{w} \text{ ”}]}$. Suppose, for a contradiction, that $\vec{w} \notin S$. Then $f(\vec{w}) = 0$, so $h(\vec{w}, k) = 0$ for all k. Therefore

$$\text{for all } k\ ,\ \ \mathcal{N} \vdash \neg J^{[\vec{x} \to \text{“ } \vec{w} \text{ ”}]\ [x_{n+1} \to \text{“}k\text{”}]} .$$

Thus $\neg J^{[\vec{x} \to \text{“ } \vec{w} \text{ ”}][x_{n+1} \to \text{“}k\text{”}]}$ is true in $\mathbf{N}$ for all $k \in \mathbf{N}$. Taking $F = \neg J^{[\vec{x} \to \text{“ } \vec{w} \text{ ”}]}$ and $x = x_{n+1}$, we have established that $F^{[x \to \text{“}k\text{”}]}$ is true in $\mathbf{N}$ for all k, and so is $\neg\forall x F$. This contradiction completes the proof.

On page 202 of [**LM**], a claim as follows is made:

The naming relation is definable in $1^{\underline{\text{st}}}$order.

This claim, as we admitted there, is the one large gap in the description given of an elegant proof by Boolos of Gödel's incompleteness theorem. That proof is based on Berry's paradox, which refers to a linguistic paradox concerning the "smallest natural number which cannot be described in less than x words", with x chosen to make sure it is larger than the number of words in quotes.

To see that the naming relation is definable, we'll argue informally here that it is semi-decidable, and appeal to the theorem just proved. Formalizing that argument is easy, using the material in the next section. (In that section, we concentrate rather on the classical proof of Gödel, based on his modification of the liar paradox which refers to derivability rather than truth.)

The naming relation was defined in [**LM**] by

$$NAME(v, w) \iff v \text{ is named by a formula of weight } w .$$

The weight of a formula was given in a way which is obviously computable in the informal sense. A natural number v being named by a formula F with respect to a decidable set Γ of formulas in $1^{\underline{\text{st}}}$order number theory was defined to mean

$$\Gamma \vdash \forall x(F \leftrightarrow x \approx \text{“}v\text{”}) .$$

We need to see that the set of (v, w) for which $NAME(v, w)$ holds is recursively enumerable, i.e. to give an informal algorithm for listing the set. But the set $\Gamma^{\vdash}$ is certainly listable, and so it is just a matter of going through that list and eliminating most of its entries, namely those which don't have the form, for some v and F, as given on the right-hand side of the display just above. And replace each formula which isn't eliminated by the pair (v, w), where w is the weight of F, producing the required list.

V-3: The converse, and using the proof system as a 'computer'.

The converse of Theorem **V-1.3** (that *any representable function is recursive*) is true for total functions, but not quite true in general. A representable function is one which, when restricted to any recursively enumerable subset of its domain, becomes recursive. This follows from Church's Thesis, since, as explained in the next few paragraphs, the proof system would give an (outrageously inefficient!) algorithm for computing some extension of the function to a possibly larger $\mathcal{RE}$ domain than it's own domain (which may very well not be $\mathcal{RE}$).

To calculate $g(\vec{v})$, given that G represents g, one first searches an 'algorithmic listing' of $\mathcal{N}^{\vdash}$ for one of the two formulae

$$x_0 \approx 0 \leftrightarrow G^{[\vec{x}\rightarrow\text{“ }\vec{v}\text{ ”}]} \qquad \text{or} \qquad G^{[\vec{x}\rightarrow\text{“ }\vec{v}\text{ ”}]} \rightarrow \neg\, x_0 \approx 0 \quad .$$

One of them will turn up eventually, at least for those $\vec{v}$ in the domain of g. If it's the second one, start a search for

$$x_0 \approx \text{“}1_N\text{”} \leftrightarrow G^{[\vec{x}\rightarrow\text{“ }\vec{v}\text{ ”}]} \qquad \text{or} \qquad G^{[\vec{x}\rightarrow\text{“ }\vec{v}\text{ ”}]} \rightarrow \neg\, x_0 \approx \text{“}1_N\text{”} \quad .$$

Eventually, once it's the first one, i.e.

$$\mathcal{N} \;\vdash\; x_0 \approx \text{“}\ell\text{”} \leftrightarrow G^{[\vec{x}\rightarrow\text{“ }\vec{v}\text{ ”}]} \quad ,$$

one has calculated that $g(\vec{v}) = \ell$. [The eventuality just above can only possibly fail for $\vec{v} \notin \text{Domain}(g)$.]

This depends on believing the set $\mathcal{N}$ to be consistent. (But anyone disbelieving that has more serious problems to deal with!) Consistency is used in proving that, if $\mathcal{N} \;\vdash\; x_0 \approx \text{“}\ell\text{”} \leftrightarrow G^{[\vec{x}\rightarrow\text{“ }\vec{v}\text{ ”}]}$, then, for all $i \neq \ell$, we have both

$$\mathcal{N} \;\nvdash\; x_0 \approx \text{“}i\text{”} \leftrightarrow G^{[\vec{x}\rightarrow\text{“ }\vec{v}\text{ ”}]} \quad \text{and} \quad \mathcal{N} \;\vdash\; G^{[\vec{x}\rightarrow\text{“ }\vec{v}\text{ ”}]} \rightarrow \neg\, x_0 \approx \text{“}i\text{”} \quad .$$

The left-hand claim holds since $\mathcal{N} \;\nvdash\; \text{“}i\text{”} \approx \text{“}\ell\text{”}$, by consistency, because we know that $\mathcal{N} \;\vdash\; \neg\, \text{“}i\text{”} \approx \text{“}\ell\text{”}$, by (III) in the list of $\mathcal{N}$-specific derivations.

The latter is also used to see that $\mathcal{N} \;\vdash\; x_0 \approx \text{“}\ell\text{”} \;\rightarrow \neg\, x_0 \approx \text{“}i\text{”}$, which with hypothetical syllogism, takes care of the other claim. The claimed derivation just above may be achieved by first writing down derivations for $\neg\, \text{“}i\text{”} \approx \text{“}\ell\text{”}$, and for

$$x_0 \approx \text{“}\ell\text{”} \;\wedge\; x_0 \approx \text{“}i\text{”} \;\longrightarrow\; \text{“}i\text{”} \approx \text{“}\ell\text{”} \quad ,$$

then applying the contrapositive to get the line

$$\neg(x_0 \approx \text{“}\ell\text{”} \wedge x_0 \approx \text{“}i\text{”}) \quad ,$$

which, up to adding a double negation, is exactly what we want, in view of the definition of $\rightarrow$.

With the work using Gödel numbering etc. in Sections IV and VI, the appeal to Church's Thesis can be eliminated, giving a *mathematical* proof of this approximate converse.

It is interesting to see how the "infinite loop" can possibly happen here when the input $\vec{v}$ is not in the domain of g. Either the formula $G^{[\vec{x}\mapsto\text{“ }\vec{v}\text{ ”}]} \to \neg x_0 \approx$ "i" keeps getting found in the list $\mathcal{N}^{\vdash}$ every time (for every i—which does seem more like a 'slow march to infinity' than an infinite loop), or else, for some i, neither that formula, nor $x_0 \approx$ "i" $\leftrightarrow G^{[\vec{x}\mapsto\text{“ }\vec{v}\text{ ”}]}$ ever turns up, and the search again goes on 'forever'.

Any decent proof system for $1^{\underline{\text{st}}}$order logic can be regarded as the basis for yet another definition of the word computable. In the sketch above, we haven't quite captured this idea completely. Using the ideas above, plus those of the previous subsection, we can conceive the following analogies.

Corresponding to a 'computer', to the 'hardware', to the general idea of a Turing machine, or to the quasi-physical notions of bins and a mechanism for carrying out **BTEN** commands, we shall place the language of $1^{\underline{\text{st}}}$order number theory, the fixing of some particular proof system for $1^{\underline{\text{st}}}$order logic (which is complete and decidable in a suitable sense), and the set $\mathcal{N}$ of very simple sentences which are obviously true in $\mathbf{N}$.

Corresponding to a program for a computer, or a particular abstract specification of a Turing machine, or to a **BTEN** command, will simply be a formula J in the language of $1^{\underline{\text{st}}}$order number theory.

'Input' will be, as usual, any n-tuple of natural numbers, $\vec{v}$, just as with **BTEN**-computability.

Definition. The function $f : D \to \mathbf{N}$ of n variables which such a formula J computes is defined as follows:
(i) $D \subset \mathbf{N}^n$ is defined to be the set of those $\vec{v}$ for which there exists an $\ell \in \mathbf{N}$ such that

$$\mathcal{N} \ \vdash \ x_0 \approx \text{“}\ell\text{”} \leftrightarrow J^{[\vec{x}\mapsto\text{“ }\vec{v}\text{ ”}]} \ ,$$

(ii) It is elementary to see (from consistency of $\mathcal{N}$) that there will be at most one ℓ as in (i), once $\vec{v}$ and J have been fixed. It is also rather easy to see informally that D is recursively enumerable. As with the 'Boolos discussion' ending the previous subsection, it's just a matter of listing $\mathcal{N}^{\vdash}$ and comparing.

(iii) Then the formula for the function f is $f(\vec{v}) := \ell$.

Finally say that a function $f : D \to \mathbf{N}$ is $1^{\underline{\text{st}}}$**order computable** if and only if there is a formula J such that f is the function of n variables which J computes.

The comments in (ii) just above together with the initial four paragraphs of this subsection give us an argument from Church's thesis that being $1^{\underline{\text{st}}}$order computable implies being recursive.

We prove the converse in the following, depending on the innocuous assumption about the ω-consistency of $\mathcal{N}$. It is hardly surprising that we need some such assumption. For example, were we all living in a fool's paradise, with $\mathcal{N}$ not even consistent, it is clear immediately from the definition that *every total* function would be $1^{\underline{\text{st}}}$order computable (but no others). However, the theorem below tells most of us to be confident that the $1^{\underline{\text{st}}}$order

computable functions are precisely the (partial) recursive functions. So the definition above is acceptable.

Theorem V-3.1. *If a function is (partial) recursive, then it is* $1^{\underline{\text{st}}}$*order computable, assuming that* $\mathcal{N}$ *is* ω*-consistent.*

Proof. Let $f : D \to \mathbf{N}$ be recursive. By Theorem **V-1.3**, choose a formula G such that

$$\vec{v} \in D \implies \mathcal{N} \vdash x_0 \approx \text{“}f(\vec{v})\text{”} \leftrightarrow G^{[\vec{x}\to\text{“ }\vec{v}\text{ ”}]} .$$

Now D is recursively enumerable, so by Theorem **V-2.2**, there is a formula H such that

$$\vec{v} \in D \iff \mathcal{N} \vdash H^{[\vec{x}\to\text{“ }\vec{v}\text{ ”}]} .$$

Because $\vec{x}$ only involves x_i for $i > 0$, we get $(\forall x_0 H)^{[\vec{x}\to\text{“ }\vec{v}\text{ ”}]} = \forall x_0 (H^{[\vec{x}\to\text{“ }\vec{v}\text{ ”}]})$. Combined with the fact that $\Gamma \vdash F$ if and only if $\Gamma \vdash \forall x_0 F$, we see that H may be replaced by $\forall x_0 H$ if necessary. Thus it may be assumed that H has no free occurrences of the variable x_0.

Now take $J := G \wedge H$. We shall prove that, for each $\vec{v} \in \mathbf{N}^n$,

$$\vec{v} \in D \iff \text{there exists an } \ell \in \mathbf{N} \text{ such that } \mathcal{N} \vdash x_0 \approx \text{“}\ell\text{”} \leftrightarrow J^{[\vec{x}\to\text{“ }\vec{v}\text{ ”}]} ;$$

and that, when $\vec{v} \in D$, that ℓ is $f(\vec{v})$. This will complete the proof that f is $1^{\underline{\text{st}}}$order computable.

To prove $\Longleftarrow$, given $\vec{v} \in \mathbf{N}^n$, fix some $\ell \in \mathbf{N}$ so that

$$\mathcal{N} \vdash x_0 \approx \text{“}\ell\text{”} \leftrightarrow J^{[\vec{x}\to\text{“ }\vec{v}\text{ ”}]} .$$

Since $\Gamma \vdash F \Rightarrow \Gamma \vdash \forall x_0 F \Rightarrow \Gamma \vdash F^{[x_0 \to t]}$ for any term t, take t to be “ℓ” in the last display, and use only $\to$, not $\leftrightarrow$, to get

$$\mathcal{N} \vdash \text{“}\ell\text{”} \approx \text{“}\ell\text{”} \to J^{[\vec{x}\to\text{“ }\vec{v}\text{ ”}][x_0\to\text{“}\ell\text{”}]} .$$

But $\vdash$ “ℓ” $\approx$ “ℓ”, so we see that

$$\mathcal{N} \vdash J^{[\vec{x}\to\text{“ }\vec{v}\text{ ”}][x_0\to\text{“}\ell\text{”}]} = G^{[\vec{x}\to\text{“ }\vec{v}\text{ ”}][x_0\to\text{“}\ell\text{”}]} \wedge H^{[\vec{x}\to\text{“ }\vec{v}\text{ ”}]} .$$

And so $\mathcal{N} \vdash H^{[\vec{x}\to\text{“ }\vec{v}\text{ ”}]}$, giving $\vec{v} \in D$, as required.

Now given $\vec{v} \in D$, we'll be done by showing

$$\mathcal{N} \vdash x_0 \approx \text{“}f(\vec{v})\text{”} \leftrightarrow J^{[\vec{x}\to\text{“ }\vec{v}\text{ ”}]} .$$

Divide this $\leftrightarrow$ formula as a conjunction of a $\to$, and a $\leftarrow$.

For the first, we note that there is a derivation from $\mathcal{N}$ of both

$$x_0 \approx \text{“}f(\vec{v})\text{”} \to G^{[\vec{x}\to\text{“ }\vec{v}\text{ ”}]} \quad \text{and of} \quad x_0 \approx \text{“}f(\vec{v})\text{”} \to H^{[\vec{x}\to\text{“ }\vec{v}\text{ ”}]} .$$

These are clear from the choices originally made of the formulae G and H. So we just use the 'propositional fact' that $\Gamma \vdash A \to B$ and $\Gamma \vdash A \to C$ certainly implies that $\Gamma \vdash A \to B \wedge C$.

For the reverse arrow, we actually have

$$\mathcal{N} \ \vdash \ G^{[\vec{x} \mapsto \text{“ } \vec{v} \text{ ”}]} \to x_0 \approx \text{“}f(\vec{v})\text{”} \quad ,$$

which does it, since $\Gamma \vdash A \to C$ certainly implies that $\Gamma \vdash A \wedge B \to C$.

Assuming that, for reasons other than computability theory, a person had already learned the basic syntax of $1^{\underline{\text{st}}}$order languages, or at least that of $1^{\underline{\text{st}}}$order number theory (reasons which certainly do exist in abundance!), it's unlikely that there could be a shorter, simpler characterization than the one given here of this fundamental idea:—computable. It does seem a topic worth including for aesthetic reasons in a logic text, if not in one purely on recursion theory/theory of computing. After all, an algorithm here is simply a formula from $1^{\underline{\text{st}}}$order number theory in disguise.

VI : The Algorithms for the Gödel, Church and Tarski Proofs

To complete the 'rigorization' of these proofs from Appendix L of [**LM**], we must replace the informal algorithms in each by an algorithm in one of the senses ($\mathcal{RC}$ or $\mathcal{BC}$) of our present (mathematically precise) formulations.

First, we define, and prove to be primitive recursive, three functions used within those algorithms, and several others needed to achieve this. This work is much like that in Subsection IV-3, a bit tedious but not subtle at all. In the next section, there are some illustrations and examples of how this could alternatively be done by writing **BTEN** algorithms to compute the functions.

Then for the algorithms themselves, we leave recursiveness behind, and use the **BTEN** language. This translation from informal algorithms to **BTEN** commands is extremely easy, and perhaps a good illustration of how the $\mathcal{BC}$ formulation of computability can be applied. Indeed, the $\mathcal{BC}$ formulation makes the same thing quite easy for just about any of the informal applications of Church's thesis. Such applications are quite plentiful in logic texts such as [**H**], later to be replaced by (recursive or register machine) mathematical versions. This formalization can sometimes be more painful than it is here using the **BTEN** language.

So, besides completing our main motivating project for this work (rigorizing the last part of the text [**LM**]), this section also provides an illustration of the usefulness of having more than one formulation of computability. Kleene's theorem and the Halting Problem material from Section IV are other good illustrations. Of course one does need to have shown the equivalence of the different formulations.

VI-1. Being a proof number is recursive.

The first job is to prove to be recursive a couple of partial functions related to substitution in $1^{\underline{\text{st}}}$order logic. Define

$$SNT_1(a,b) \ := \ \#((\text{term}\#a)^{[x_1 \mapsto \text{term}\#b]}) \quad .$$

The right-hand side is the code of the term obtained by substituting term$\#b$ for x_1 in term$\#a$. So we are defining a function

$$SNT_1 : D \to \mathbf{N} \ \text{ where } \ D = \{ \ (a,b) \in \mathbf{N}^2 \ : \ TRM(a) = 1 = TRM(b) \ \} \quad .$$

We shall be using many facts from Appendix A : for example that TRM is a recursive relation, where $TRM(a) = 1$, usually just written $TRM(a)$, means that a is the code number of some term in the language of 1$^{\underline{\text{st}}}$order number theory.

As illustrated in much of Section IV, we turn the proof of recursiveness of SNT_1 into the same question for a (total) relation in one extra variable, namely

$$TSN_1(a,b,c) \ := \ \begin{cases} 1 & \text{if } c = SNT_1(a,b) \ ; \\ 0 & \text{otherwise} \ . \end{cases}$$

This suffices, as usual, since

$$SNT_1(a,b) \ = \ \min\{ \ c \ : \ TSN_1(a,b,c) = 1 \ \} \ .$$

But now, from the definitions above,

$$\underline{TSN_1(a,b,c)} \qquad \Longleftrightarrow \qquad TRM(a), TRM(b) \text{ and } TRM(c) \ ; \text{ and one of } :$$

$$CLV(a,1) = 0 \ \text{ and either } (I) \quad CLV(c,3) = 2 \quad \text{and } b = c \ ,$$

$$\text{or } (II) \quad CLV(c,3) \neq 2 \quad \text{and } a = c \quad ;$$

[Here term#a is x_1 in case (I), and
is 0, or 1, or x_j with $j \neq 1$, in case (II).]

$$\text{or} \qquad CLV(a,1) = 1 = CLV(c,1) \ \text{ and}$$

$$TSN_1(CLV(a,2), b, CLV(c,2)) \ \text{ and } \ TSN_1(CLV(a,3), b, CLV(c,3)) \quad ;$$

[Here term#a has the form $s + t$.]

$$\text{or} \qquad CLV(a,1) = 2 = CLV(c,1) \ \text{ and}$$

$$TSN_1(CLV(a,2), b, CLV(c,2)) \ \text{ and } \ TSN_1(CLV(a,3), b, CLV(c,3))$$

[Here term#a has the form $s \times t$.]

By general results in Appendix A, this shows TSN_1 to be recursive, since we proved TRM and CLV to be recursive in Appendix A. And now we also know SNT_1 to be recursive.

Next define

$$SNF_1(a,b) \ := \ \#((\text{formula}\#a)^{[x_1 \to \text{term}\#b]}) \ .$$

The right-hand side is the code of the formula obtained by substituting term#b for all free occurrences of x_1 in formula#a. So we are defining a function

$$SNF_1 : D \to \mathbf{N} \ \text{ where } \ D = \{ \ (a,b) \in \mathbf{N}^2 \ : \ FRM(a) \ \text{ and } \ TRM(b) \ \} \ .$$

As just above, we turn the proof of recursiveness of SNF_1 into the same question for a (total) relation in one extra variable, namely

$$FSN_1(a,b,c) \ := \ \begin{cases} 1 & \text{if } c = SNF_1(a,b) \ ; \\ 0 & \text{otherwise} \ . \end{cases}$$

This works, as before, because

$$SNF_1(a,b) \ = \ \min\{ \ c \ : \ FSN_1(a,b,c) = 1 \ \} \ .$$

From the definitions above,

$$\underline{FSN_1(a,b,c)} \qquad \Longleftrightarrow \qquad FRM(a) \ , \ TRM(b) \text{ and } FRM(c) \ ; \text{ and one of } :$$

$$CLV(a,1) = 1 = CLV(c,1) \ \text{ and}$$

$$TSN_1(CLV(a,2), b, CLV(c,2)) \ \text{ and } \ TSN_1(CLV(a,3), b, CLV(c,3)) \ ;$$

[Here formula#a has the form $s \approx t$,
with $CLV(a,2) = \#s$ and $CLV(a,3) = \#t$.]

$$\text{or} \qquad CLV(a,1) = 2 = CLV(c,1) \ \text{ and}$$

$$TSN_1(CLV(a,2), b, CLV(c,2)) \ \text{ and } \ TSN_1(CLV(a,3), b, CLV(c,3)) \ ;$$

[Here formula#a is $s < t$.]

$$\text{or} \qquad CLV(a,1) = 3 = CLV(c,1) \ \text{ and } \ FSN_1(CLV(a,2), b, CLV(c,2)) \ ;$$

[Here formula#a has the form $\neg G$ with $CLV(a,2) = \#G$,
since $\#(\neg G) =< 3, \#G, \#G >$.]

$$\text{or} \qquad CLV(a,1) = 4 = CLV(c,1) \ \text{ and}$$

$$FSN_1(CLV(a,2), b, CLV(c,2)) \ \text{ and } \ FSN_1(CLV(a,3), b, CLV(c,3)) \ ;$$

[Here formula$\#a$ has the form $G \wedge H$, with
$CLV(a,2) = \#G$ and $CLV(a,3) = \#H$.]

$$\text{or} \qquad CLV(a,1) = 5 = CLV(c,1) \ \text{ and } \ CLV(a,2) = CLV(c,2) \neq 1 \ \text{ and}$$

$$FSN_1(CLV(a,3), b, CLV(c,3)) \ ;$$

[Here formula$\#a$ has the form $\forall x_j G$ with $j \neq 1$,
so $CLV(a,2) = j$ and $CLV(a,3) = \#G$.]

$$\text{or} \qquad CLV(a,1) = 5 = CLV(c,1) \ \text{ and } \ CLV(a,2) = 1 \ \text{ and } \ a = c \ .$$

[Here formula$\#a$ has the form $\forall x_1 G$, and equals formula$\#c$.]

This shows FSN_1 to be recursive, since we also proved FRM to be recursive in Appendix A. And now we know that SNF_1 is recursive.

Because the last line of a derivation is the 'conclusion', we need primitive recursivity of the following, 'last line', relation :

$$LL \ : \ \mathbf{N}^2 \ \rightarrow \ \mathbf{N} \ ,$$

where $LL(a,b) = 1$ holds iff b is the code of a tuple and a is the last entry in that tuple.

So

$$\underline{LL(a,b)} \qquad \Longleftrightarrow \qquad CDE(b) \ \text{ and } \ CLV(b, CLV(b,0)) = a \ .$$

This shows LL to be primitive recursive.

Now for the big one. Suppose that $\mathcal{A}$ is a decidable set of formulae in the language of $1^{\underline{\text{st}}}$order number theory. That means there is a **recursive** relation

$$MBR_{\mathcal{A}} \ : \ \mathbf{N} \ \rightarrow \ \mathbf{N} \ ,$$

such that

$$MBR_{\mathcal{A}}(b) \Longleftrightarrow b \text{ is the code of a formula, i.e. } FRM(b), \text{ and}$$
$$\text{formula}\#b \text{ is a member of the set } \mathcal{A} \ .$$

We wish to prove that

$$PF_{\mathcal{A}} \ : \ \mathbf{N} \ \rightarrow \ \mathbf{N} \ \ ,$$

is also a recursive relation, defined by

$$PF_{\mathcal{A}}(c) \Longleftrightarrow c \text{ is the code of a derivation from } \mathcal{A} \ ,$$

where we define

$$\#_{SEQ}(F_1, F_2, \cdots, F_\ell) \ := \ \ < \#_{FRM} F_1 \ , \ \#_{FRM} F_2 \ , \ \cdots \ , \ \#_{FRM} F_\ell > \quad .$$

This last display assigns a code number to every finite sequence of formulae from 1$^{\underline{\text{st}}}$order number theory. It is evident that different sequences get different codes.

Below, the proof system which will be employed is the one for " $\vdash$ " described starting on page 241 of **[LM]**, as opposed to the one for " $\vdash^*$ ". Actually the proof system used will be slightly modified, by altering one axiom slightly, and by using a simpler set of equality axioms. Details of these modifications will be given below at the appropriate point.

Now define a relation $AX \ : \ \mathbf{N} \rightarrow \mathbf{N}$ by

$AX(b) \Longleftrightarrow FRM(b)$ and formula$\#b$ is a logical axiom in the proof system used.

Also, for each $k \geq 2$, define a relation $ROI_k \ : \ \mathbf{N}^k \rightarrow \mathbf{N}$ by

$$ROI_k(b_1, \cdots, b_k) \Longleftrightarrow FRM(b_1) \text{ and } \cdots \text{ and } FRM(b_k) \text{ and}$$

formula$\#b_k$ is immediately inferrable, using some rule of inference (in the proof system used) from one or two of (formula$\#b_1$), $\cdots$, (formula$\#b_{k-1}$) .

Then we clearly have

$$PF_{\mathcal{A}}(c) \Longleftrightarrow CDE(c) \ \text{ and } \ \forall k \text{ with } 1 \leq k \leq CLV(c, 0) \ ,$$
$$\text{we have one of } : \ \ MBR_{\mathcal{A}}(CLV(c,k)) \text{ or } AX(CLV(c,k)) \text{ or}$$
$$ROI_k(CLV(c,1), CLV(c,2), \cdots, CLV(c,k)) \ .$$

Thus $PF_{\mathcal{A}}$ is seen to be recursive, as soon as we have proved that AX and all the ROI_k are.

Because all the other ingredients used are now known to be primitive recursive, the argument which we flesh out below also shows that

$$MBR_{\mathcal{A}} \text{ is } \mathcal{PRC} \implies PF_{\mathcal{A}} \text{ is } \mathcal{PRC} \ .$$

This won't be mentioned specifically below.

VI-2. Dealing with rules of inference.

We shall deal with each of the rules of inference separately, in all cases producing a relation as in the display just below. Once those five relations have each been proved recursive, the desired result (that ROI_k is recursive) will follow from the fact that

$$ROI_k(b_1, \cdots, b_k) \Longleftrightarrow MP_k(b_1, \cdots, b_k) \text{ or } NEG_k(b_1, \cdots, b_k) \text{ or}$$

$$COM_k(b_1, \cdots, b_k) \text{ or } ASS_k(b_1, \cdots, b_k) \text{ or } GEN_k(b_1, \cdots, b_k) \ .$$

As for the (MP) (modus ponens) rule of inference, define the relation $MP_3' : \mathbf{N}^3 \to \mathbf{N}$ by

$$MP_3'(a,b,c) \iff FRM(a)\ ,\ FRM(b)\ ,\ FRM(c)\ ,\ \text{and}\ \ \text{formula}\#b \ \text{ is}$$

$$(\text{formula}\#a \to \text{formula}\#c) \ := \ \neg(\text{formula}\#a \wedge \neg \text{formula}\#c) \ .$$

Define

$$MP_3(a,b,c) \iff MP_3'(a,b,c) \text{ or } MP_3'(b,a,c) \ .$$

Define, for $k \geq 3$, the relation

$$MP_k(b_1, b_2, \cdots, b_k) \iff \text{for some } i < j < k \text{ we have } MP_3(b_i, b_j, b_k) \ .$$

Thus all the MP_k will be recursive as long as MP_3' is, and the latter is proved by the following :

$$\underline{MP_3'(a,b,c)} \qquad \Longleftrightarrow \qquad FRM(a)\ ,\ FRM(b)\ ,\ FRM(c)\ ,$$

$$CLV(b,1)=3\ ,\ CLV(CLV(b,2),2)=a\ ,\ CLV(CLV(b,2),1)=4\ ,$$

$$CLV(CLV(CLV(b,2),3),1)=3 \quad \text{and} \quad CLV(CLV(CLV(b,2),3),2)=c\ .$$

These equalities arise from the fact that

$$\#(\neg(\text{formula}\#a\wedge\neg\text{formula}\#c) =< 3,< 4,a,< 3,c,c >>,< 4,a,< 3,c,c >>>\ .$$

There is a way to check whether one has remembered to write down all the needed equalities in this type of analysis. The reader might wish to do the same for some of the later ones. Actually, for this it might have been better if we had defined the code of a negation as the code of a pair, rather than triple, erasing the redundancy. That is, temporarily change $\#\neg(\text{formula}\#c)$ to $< 3,c >$. Then the right-hand side of the display just above would have become $< 3,< 4,a,< 3,c >>>$. Hence the five equalities earlier, one for each of the five numbers just written down, with that number appearing as the right-hand side of the equality.

As for the $(\neg\neg)$ rule of inference, define a relation $NEG_2' : \mathbf{N}^2 \to \mathbf{N}$ by

$NEG_2'(a,b) \iff FRM(a)\,,\ FRM(b)\,,$ and formula$\#b$ is obtainable from formula$\#a$ by double-negating one of its subformulae.

Define

$$NEG_2(a,b) \iff NEG_2'(a,b) \text{ or } NEG_2'(b,a)\ .$$

Define, for $k \geq 2$, the relation

$$NEG_k(b_1,b_2,\cdots,b_k) \iff \text{for some } i<k \text{ we have } NEG_2(b_i,b_k)\ .$$

Thus all the NEG_k will be recursive as long as NEG_2' is, and the latter is proved by the following :

$$\underline{NEG_2'(a,b)} \qquad \Longleftrightarrow \qquad FRM(a) \ , \ \ FRM(b) \ , \ \text{and}$$

$\underline{\text{either}} \ \ CLV(b,1) = 3 \ , \ \ CLV(CLV(b,2),1) = 3 \ , \ \text{and} \ CLV(CLV(b,2),2) = a \ ,$

[Here formula$\#b$ is actually $\neg\neg$formula$\#a$.
So $CLV(b,2) = \#(\neg\text{formula}\#a) \ , \ b =< 3, < 3, a, a >, < 3, a, a >>$,
giving the conditions above.
In all the cases below, the $\neg\neg$ applies
to a *proper* subformula of formula$\#a$
to produce formula$\#b$.]

$$\underline{\text{or}} \ \ CLV(b,1) = CLV(a,1) = k \ \text{say, and one of} \ :$$

$$k = 3 \quad \text{and} \quad NEG_2'(CLV(a,2), CLV(b,2)) \ \ \text{or}$$

$k = 4$ and $CLV(a,2) = CLV(b,2)$ and $NEG_2'(CLV(a,3), CLV(b,3))$ or

$k = 4$ and $CLV(a,3) = CLV(b,3)$ and $NEG_2'(CLV(a,2), CLV(b,2))$ or

$k = 5$ and $CLV(a,2) = CLV(b,2)$ and $NEG_2'(CLV(a,3), CLV(b,3))$.

For the (COM) and (ASS) rules of inference, we shall leave some work for the reader to do. Firstly, ponder the last several lines just above to see what's going on, and convince yourself that it's correct. Then deal with the above two rules by analogy with the previous few pages. That is, produce relations COM_k and ASS_k, and probably ASS_2' , and prove them recursive.

Finally, for the rule (GEN), define

$$GEN_2(a,b) \iff FRM(a) \text{ and } FRM(b) \text{ and, for some } j \ ,$$

$$\text{formula}\#b \ \text{ is } \ \forall x_j(\text{formula}\#a) \ .$$

Then GEN_2 is recursive, since the last condition amounts to

$$CLV(b,1) = 5 \ \text{ and } \ CLV(b,3) = a \ .$$

Of course, we define $GEN_k(b_1, \cdots, b_k)$ to mean that, for some $i < k$, we have $GEN_2(b_i, b_k)$. And so GEN_k is recursive.

This completes all that is needed to show ROI_k to be recursive.

VI-3. Dealing with the axioms.

This is similar to the last subsection, but slightly messier. We'll consider seven axiom schema, reducing the equality axioms to two, by use of Exercise 7.11, page 280 in **[LM]**.

Thus we shall have

$$AX(b) \Longleftrightarrow AX_1(b) \text{ or } AX_2(b) \text{ or } \cdots \text{ or } AX_7(b) \ ,$$

and the job (showing AX to be recursive) reduces to showing that each AX_j is recursive.

The relation AX_1 corresponds to the axiom schema $F \to F \wedge F$. Thus the definition is

$AX_1(b) \iff FRM(b)$ and, for some formula F, formula$\#b$ is $F \to F \wedge F$.

Here we again pay a small price for the otherwise economical convention of regarding "$\to$" as an abbreviation, since

$$F \to F \wedge F \text{ is really } \neg(F \wedge \neg(F \wedge F)) \ .$$

The code of the latter is

$$< 3 \ , \ \#(F \wedge \neg(F \wedge F)) \ , \ \#(F \wedge \neg(F \wedge F)) > \ = \ < 3, a, a > \ ,$$

where

$$a \ = \ < 4 \ , \#F \ , \#\neg(F \wedge F) > \ = \ < 4 \ , \#F \ , \ < 3 \ , \#(F \wedge F) \ , \#(F \wedge F) >> \ =$$

$$< 4 \ , \ \#F \ , \ < 3 \ , \ < 4 \ , \ \#F \ , \ \#F > \ , \ < 4 \ , \ \#F \ , \ \#F >>> \ .$$

Thus

$\underline{AX_1(b)} \iff FRM(b) \ , \ CLV(b,1) = 3 \ ,$ and, with $a = CLV(b,2)$, we have

$$CLV(a,1) = 4 \ , \text{ and, with } c = CLV(a,3), \text{ we have}$$
$$CLV(c,1) = 3 \ , \ CLV(CLV(c,2),1) = 4 \text{ and}$$
$$CLV(CLV(c,2),2) = CLV(a,2) = CLV(CLV(c,2),3) \ .$$

This shows AX_1 to be recursive.

We'll leave to the reader as an exercise the analogous work for relations AX_2 and AX_3 , corresponding respectively to axiom schemes $F \wedge G \to F$ and $(F \to G) \to (F \wedge H \to G \wedge H)$.

Before doing the final four axiom schema (which are particular to $1^{\underline{\text{st}}}$order logic, as opposed to propositional logic), we need recursivity of three more functions, related to the free occurrence of variables and substitutability of terms for them.

Define

$TF(j,a) \iff a$ is a code of a term in which the variable x_j does not occur.

Thus

$$\underline{TF(j,a)} \iff TRM(a) \text{ and one of}$$
$$CLV(a,1) = 0 \text{ and } CLV(a,3) \neq j+1 \text{ ; or}$$
$$CLV(a,1) > 0 \text{ , } TF(j,CLV(a,2)) \text{ and } TF(j,CLV(a,3)) \text{ .}$$

This shows TF to be recursive.

Define

$FF(j,b) \iff b$ is a code of a formula in which no occurrence of the variable x_j is free.

That is, in **[LM]**-jargon, "formula#b is FOFOO x_j" .

Thus

$$\underline{FF(j,b)} \iff FRM(b) \text{ and one of}$$
$$CLV(b,1) = 1 \text{ , } TF(j,CLV(b,2)) \text{ and } TF(j,CLV(b,3)) \text{ ; or}$$
$$CLV(b,1) = 2 \text{ , } TF(j,CLV(b,2)) \text{ and } TF(j,CLV(b,3)) \text{ ; or}$$
$$CLV(b,1) = 3 \text{ and } FF(j,CLV(b,2)) \text{ ; or}$$
$$CLV(b,1) = 4 \text{ , } FF(j,CLV(b,2)) \text{ , and } FF(j,CLV(b,3)) \text{ ; or}$$
$$CLV(b,1) = 5 \text{ and } CLV(b,2) = j \text{ ; or}$$
$$CLV(b,1) = 5 \text{ , } CLV(b,2) \neq j \text{ , and } FF(j,CLV(b,3)) \text{ .}$$

This shows FF to be recursive.

Define

$$SUB(j,a,b) \iff TRM(a) \text{ and } FRM(b) \text{ and term\#}a \text{ is}$$

$$\text{substitutable for } x_j \text{ in formula\#}b \ .$$

Recall that the latter means that all variables in term#a "remain" free after the substitution (for *free* occurrences of x_j in formula#b, of course).

Thus

$$\underline{SUB(j,a,b)} \iff TRM(a) \text{ and } FRM(b) \text{ and one of}$$

$$CLV(b,1) \leq 2 \ ; \text{ or}$$

$$CLV(b,1) = 3 \text{ and } SUB(j,a,CLV(b,2)) \ ; \text{ or}$$

$$CLV(b,1) = 4 \ , \ SUB(j,a,CLV(b,2)) \ , \text{ and } SUB(j,a,CLV(b,3)) \ ; \text{ or}$$

$$CLV(b,1) = 5 \text{ and } CLV(b,2) = j \ ; \text{ or}$$

$$CLV(b,1) = 5 \ , \ CLV(b,2) \neq j \ , \ FF(j,b) = 0 \neq 1!$$

$$SUB(j,a,CLV(b,3)) \text{ and } TF(CLV(b,2),a) \ ; \text{ or}$$

$$CLV(b,1) = 5 \ , \ CLV(b,2) \neq j \ , \text{ and } FF(j,b) \ .$$

With F =formula#b, the last three conditions correspond, respectively, to :
when F has the form $\forall x_j G$, so there are no free x_j to worry about ;
when F has the form $\forall x_k G$, $k \neq j$, G has free x_j
and term#a is substitutable for x_j in G , but term#a has no x_k ;
when F has the form $\forall x_k G$, with $k \neq j$, but G has no free x_j .

This shows SUB to be recursive.

The relation AX_4 corresponds to the axiom schema $\forall x_j F \to F^{[x_j \to t]}$. Thus the definition is

$AX_4(b) \iff FRM(b)$ and, for some j, some term t and some formula F ,

t is substitutable for x_j in F and formula$\#b$ is $\forall x_j F \to F^{[x_j \to t]}$.

The latter formula is really $\neg(\forall x_j F \wedge \neg F^{[x_j \to t]})$, with code $< 3, p, p >$, where

$$p \;=\; \#(\forall x_j F \wedge \neg F^{[x_j \to t]}) \;=\; < 4, < 5, j, \#F >, < 3, \ell, \ell >> \quad ,$$

where

$$\ell \;=\; \#F^{[x_j \to t]} \;=\; SNF(j, \#F, \#t) \quad ,$$

where, generalizing FSN_1 and SNF_1 :

Exercise. Show that FSN and SNF are recursive, the definitions being :

$SNF(j, a, b) = k \iff FRM(a)$, $TRM(b)$ and $k = \#((\text{formula}\#a)^{[x_j \to \text{term}\#b]})$,

and

$$FSN(j, a, b, c) \;:=\; \begin{cases} 1 & \text{if } c = SNF(j, a, b) \;\; ; \\ 0 & \text{otherwise} \;\; . \end{cases}$$

Thus $SNF_1(**)$ becomes $SNF(1, **)$, and similarly for FSN_1 .
(The 'same' exercise can be done for terms, giving SNT and TNS.)

Thus we have

$\underline{AX_4(b)} \iff FRM(b)$, $CLV(b, 1) = 3$ and $CLV(b, 2) = p$; where

$CLV(p, 1) = 4$, $CLV(CLV(p, 2), 1) = 5$, $CLV(CLV(p, 3), 1) = 3$;

where, with $j = CLV(CLV(p, 2), 2)$,

$k = CLV(CLV(p, 2), 3)$,

$\ell = CLV(CLV(p, 3), 2)$, and

$c = \min\{d : FSN(j, k, d, \ell) = 1\}$,

we have $SUB(j, c, k)$.

This shows AX_4 to be recursive.

The relation AX_5 corresponds to the axiom schema (restricted to formulae F having no free occurrence of x_j) :

$$\forall x_j(F \to G) \ \to \ (F \to \forall x_j G) \ ,$$

Thus its definition is

$$AX_5(b) \iff FRM(b) \ \text{and, for some } j, \ \text{ some term } t$$

$$\text{and some } F \text{ and } G \ , \ \text{formula } F \text{ is FOFOO } x_j \ \ ,$$

$$\text{and formula}\#b \ \text{ is } \ \forall x_j(F \to G) \ \to \ (F \to \forall x_j G) \ \ .$$

The latter formula is really (unabbreviated)

$$\neg(\forall x_j \neg(F \wedge \neg G) \wedge \neg\neg(F \wedge \neg\forall x_j G)) \ \ .$$

To simplify, let us now change the proof system slightly, by removing the $\neg\neg$ in the middle of this formula. In view of the rule of inference $(\neg\neg)$, this altered proof system is immediately seen to be equivalent to the original one. So our new definition of $AX_5(b)$ replaces the last phrase above by

"..... formula$\#b$ is $\neg(\forall x_j \neg(F \wedge \neg G) \wedge (F \wedge \neg\forall x_j G))$."

This last formula has code $< 3, t, t >$, where $t =< 4, r, s >$, where

$$r \ = \ < 5, j, < 3, < 4, \#F, < 3, \#G, \#G >>, < 4, \#F, < 3, \#G, \#G >>>> \ \ ,$$

and

$$s \ = \ < 4, \#F, < 3, < 5, j, \#G >, < 5, j, \#G >>> \ \ .$$

Thus we have

$$\underline{AX_5(b)} \iff FRM(b)\ ,\ CLV(b,1)=3 \text{ and } :$$
$$CLV(CLV(b,2),1)=4\ ,$$
$$CLV(CLV(b,2),2)=r\ ,$$
$$CLV(CLV(b,2),3)=s\ ,$$

where

$$CLV(r,1)=5\ ,\ CLV(s,1)=4\ ,$$
$$CLV(CLV(r,3),1)=3\ ,\ CLV(CLV(s,3),1)=3\ ,$$
$$CLV(CLV(CLV(CLV(r,3),2),3),1)=3\ ,$$
$$CLV(CLV(CLV(s,3),2),1)=5\ ,\ CLV(CLV(CLV(r,3),2),1)=4$$
$$FF((CLV(r,2),CLV(s,2))$$
$$CLV(s,2)=CLV(CLV(CLV(r,3),2),3)$$
$$CLV(r,2)=CLV(CLV(CLV(s,3),2),2)$$
$$CLV(CLV(CLV(CLV(r,3),2),3),2)=CLV(CLV(CLV(s,3),2),3) \quad .$$

The last three equalities correspond to matching the occurrences of $\#F$, of j, and of $\#G$, respectively, in the previous expressions for r and s. The nine equalities with numbers on the right-hand side correspond to the numbers occurring in those expressions. The completeness of our set of equalities is easier to see if we temporarily redefine the code of a negation using the code of an ordered pair, and rewrite the expressions for r and s, as illustrated earlier with the analysis of MP_3' .

This shows AX_5 to be recursive.

By Ex. 7.11, page 280 in [**LM**], we can further modify the initially stated proof system by using for equality axioms : a single axiom scheme, plus one particular axiom.
The relation AX_6 corresponds to the latter, namely $x_1 \approx x_1$.
The relation AX_7 corresponds to the former, namely

$$F \wedge\ x_j \approx x_k\ \rightarrow\ F^{[x_j \to x_k]}$$

for all atomic formulae F and all j,k such that x_k doesn't occur in F .

So the first definition is

$$\underline{AX_6(b)} \iff FRM(b)\ ,\ CLV(b,1)=1\ ,\ CLV(b,2)=CLV(b,3)\ \text{ and}$$
$$CLV(CLV(b,2),1)=0\ ,$$
$$CLV(CLV(b,2),2)=1\ ,$$
$$\text{and}\ \ CLV(CLV(b,2),3)=2\ ,$$

manifestly recursive (indeed AX_6 maps all of $\mathbf{N}$ to 0, except for one number). These equations were motivated by wanting

$$AX_6(b) \iff b=\#(x_1\approx x_1)=<1,\#x_1,\#x_1>=<1,<0,1,2>,<0,1,2>>\ .$$

The final definition is

$$AX_7(b) \iff FRM(b)\ \text{ and, for some } j \text{ and } k \text{ and some terms } t \text{ and } s\ ,$$
$$x_k \text{ doesn}'\text{t occur in } t \text{ or } s, \text{ and formula} \#b \text{ is}$$
$$\text{either}\ \ s\approx t\ \wedge\ x_j\approx x_k\ \longrightarrow\ s^{[x_j\to x_k]}\approx\ t^{[x_j\to x_k]}\ ,$$
$$\text{or}\ \ s<t\ \wedge\ x_j\approx x_k\ \longrightarrow\ (s^{[x_j\to x_k]}<\ t^{[x_j\to x_k]}\ .$$

These latter two formulae are really

$$\neg((s\approx t\ \wedge\ x_j\approx x_k)\ \wedge\ \neg\ s^{[x_j\to x_k]}\approx\ t^{[x_j\to x_k]})\ ,$$
$$\text{and}\ \ \neg((s<t\ \wedge\ x_j\approx x_k)\ \wedge\ \neg\ s^{[x_j\to x_k]}<\ t^{[x_j\to x_k]})\ .$$

Their codes are, respectively $<3,p,p>$ and $<3,q,q>$, where p is

$$<4,<4,<1,\#s,\#t>,<1,\#x_j,\#x_k>>,$$
$$<3,<1,SNF(j,\#x_k,\#s),SNF(j,\#x_k,\#t)>,$$
$$<1,SNF(j,\#x_k,\#s),SNF(j,\#x_k,\#t)>>,$$

and q is the same except the 1st, 3rd, and 4th "1's" become "2's". Recall also that $\#x_j = <0\ ,\ j\ ,\ j+1>$.

Thus we can see that AX_7 is recursive, and thereby complete the proof that AX is also, and finally that $PF_{\mathcal{A}}$ is recursive, by seeing that

$\underline{AX_7(b)} \iff FRM(b)\ ,\ CLV(b,1) = 3$ and $CLV(b,2) = p$ where :

either all of the following hold

$$CLV(p,1) = 4\ ,$$

$$CLV(CLV(p,2),1) = 4\ ,$$
$$CLV(CLV(p,3),1) = 3\ ,$$
$$CLV(CLV(CLV(p,2),2),1) = 1\ , \quad (*)$$
$$CLV(CLV(CLV(p,2),3),1) = 1\ ,$$
$$CLV(CLV(CLV(p,3),2),1) = 1\ , \quad (*)$$

and, defining numbers

$$s_1 = CLV(CLV(CLV(p,2),2),2)\ ,$$

$$t_1 = CLV(CLV(CLV(p,2),2),3)\ ,$$
$$j = CLV(CLV(CLV(CLV(p,2),3),2),2)\ ,$$
$$k = CLV(CLV(CLV(CLV(p,2),3),3),2)\ ,$$

we have $TF(k,s_1)\ ,\ TF(k,t_1)$ and

$$CLV(CLV(CLV(CLV(p,2),3),2),1) = 0\ ,$$

$$CLV(CLV(CLV(CLV(p,2),3),3),1) = 0\ ,$$
$$CLV(CLV(CLV(CLV(p,2),3),2),3) = j+1\ ,$$
$$CLV(CLV(CLV(CLV(p,2),3),3),3) = k+1\ ,$$
$$CLV(CLV(CLV(p,3),2),2) = SNT(j,CTR(0,k,k+1),s_1)\ ,$$
$$CLV(CLV(CLV(p,2),3),3),3) = SNT(j,CTR(0,k,k+1),t_1)\ ;$$

or the same conditions hold, except for changing "1" to "2" on the right-hand sides in both $(*)$'s about 15 lines above .

VI-4. The BTEN Commands for the three big theorems.

Now finally we can give the formal rewrites, in the **BTEN** language, of the informal algorithms appearing in the proofs in Appendix L of [**LM**] . Despite the technicalities of Subsection IV-3 in only showing that **ATEN**-computable implies recursive, we are allowed to use the richer **BTEN** here. That was explained in Subsection IV-2.

Using the equivalence of our computability definitions, we shall denote by $C[f]$ any choice of **BTEN**-command which computes a given recursive function f , in the strong sense : as long as bins 1 to k contain $v_1, \cdots, v_k$ respectively, we end up with $f(v_1, \cdots, v_k)$ in bin 0 when (and if !) the computation terminates, whether or not bins other than 1 to k contain 0's at the start of the computation.

Accompanying all three of the **BTEN** programs below, on the right we have traced step-by-step the effects of the command on a relevant input. This is a bit like writing down the associated history to the computation. But each $C[f]$ which is used is of course not broken down into atomic subcommands, so these are very abbreviated histories. In any case they constitute semantic proofs that the programs do what is claimed. In Section **VIII** later, there is a discussion of syntactic proofs; i.e. proof systems for program correctness.

Below, the function $SYN : \mathbf{N} \to \mathbf{N}$, which is given by

$$SYN(a) = \#_{TRM}(\text{“}a\text{”}) \ ,$$

is used. We showed it to be primitive recursive in Appendix A. And so there is a **BTEN** command, denoted $C[SYN]$ below, which computes it, even a **PTEN** command.

Church's 'Undecidability of Mathematics' Theorem.

For this we had been given a set $\mathcal{A}$ of $1^{\underline{\text{st}}}$order number theory formulae, and for a contradiction, it was assumed that derivability from $\mathcal{A}$ was decidable. Since Hilbert had posed the question, formalizing this assumption uses a function (non-recursive as Church showed!) which we shall name after him.

So let

$$HIL_{\mathcal{A}} \ : \ D \longrightarrow \mathbf{N} \ ,$$

where D is the set of all formula numbers, be the 01-function given by

$$HIL_{\mathcal{A}}(b) = 1 \iff (\text{formula}\#b) \in \mathcal{A}^{\vdash} \quad , \qquad \text{that is,}$$

$\ldots \Longleftrightarrow$ there is a derivation of formula$\#b$ with $\mathcal{A}$ as the set of premisses.

The informal algorithm in the proof **[LM]**, page 395, is this :

Input $a \in \mathbf{N}$
Is a *a formula#?*
If no, output 1.
If yes, does $(\textit{formula}\#a)^{[x_1 \mapsto \text{“}a\text{”}]} \in \mathcal{A}^{\vdash}$*?* (using the assumed decidability)
If yes, output 1.
If no, output 0.

A **BTEN**-command for it is on the left, semantic verification on the right, output double-underlined:

$\underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}(0, a, \underline{0})$

$(\mathcal{B}_1 C[FRM]\mathcal{E}$;

$\underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}(FRM(a), a, - - -)$

if $x_0 \approx 0$

$\underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}(0, - - -)$

thendo $x_0 \Leftarrow 1$

$\underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}(\underline{1}, - - -) \qquad (1, a, - - -)$

elsedo $(\mathcal{B}_1 C[SYN]\mathcal{E}$;

$\underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}(SYN(a), a, - - -)$

$x_2 \Leftarrow x_0$;

$\underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}(-, a, SYN(a), - - -)$

$C[SNF_1]$;

$\underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}(SNF_1(a, SYN(a)), - - -)$

$x_1 \Leftarrow x_0$;

$\underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}(-, SNF_1(a, SYN(a)), - - -)$

$C[HIL_{\mathcal{A}}]$))

$\underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}(\underline{\underline{HIL_{\mathcal{A}}(SNF_1(a, SYN(a)))}}, - - -)$

Gödel's Incompleteness Theorem.

Here, the informal algorithm appearing in the proof ([**LM**], pages 396-7) is actually the same as the following (slight re-wording) :

Input $(a, b) \in \mathbf{N}^2$.
Is a *a formula# ?*
If no, output 1.
If yes, find (formula#$a)^{[x_1 \mapsto \text{“}a\text{”}]}$.
Is b *a proof#, and, if so,*
does proof#b *prove (formula#*$a)^{[x_1 \mapsto \text{“}a\text{”}]}$ *?* (using decidability of $\mathcal{A}$ itself)
If no, output 1.
If yes, output 0.

Denote by G the total recursive function $(b, c) \mapsto FRM(b)PF_{\mathcal{A}}(c)LL(c, b)$. Notice that $G(b, c)$ holds if and only if b is a formula number, c is a proof number (using $\mathcal{A}$ as the premiss set), and the latter proof proves the former formula.

Here is a **BTEN**-command formalizing the above informal algorithm.

$$\underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}(0, a, b, \underline{0})$$
$(\mathcal{B}_{1,2}C[FRM]\mathcal{E}$;
$$\underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}(FRM(a), a, b, ----)$$
if $x_0 \approx 0$
$$\underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}(0, ----)$$
thendo $x_0 \leftarrow 1$
$$\underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}(\underline{1}, ----) \qquad\qquad (1, a, b, ----)$$
elsedo $(\mathcal{B}_{1,2}C[SYN]\mathcal{E}$;
$$\underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}(SYN(a), a, b, -----)$$
$\mathcal{B}_2(x_2 \leftarrow x_0$; $C[SNF_1])\mathcal{E}$; $\underline{\quad\quad}(-, a, SYN(a), ----)$
$$\underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}(SNF_1(a, SYN(a)), -, b, ---)$$
$x_1 \leftarrow x_0$;
$$\underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}(-, SNF_1(a, SYN(a)), b, ---)$$
$C[G]$;
$$\underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}(G(SNF_1(a, SYN(a)), b), ---)$$
if $x_0 \approx 0$
$$\underline{\qquad\qquad\qquad\qquad\qquad\qquad}(0, ----)$$
thendo $x_0 \leftarrow 1$
$$\underline{\qquad\qquad\qquad\qquad\qquad\qquad}(\underline{1}, ----) \qquad (1, -----)$$
elsedo $x_0 \leftarrow 0$))
$$\underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}(\underline{0}, -----)$$

Tarski's 'Undefinability of Truth' Theorem.

For this theorem, the informal algorithm implicit in the proof ([**LM**], pages 398-9) is the following :

Input $(a, b) \in \mathbf{N}^2$.
Is a *a formula#* ?
If no, output 1.
If yes, find $\#((\textit{formula\#}a)^{[x_1 \mapsto \text{“}a\text{”}]})$.
Is $\#((\textit{formula\#}a)^{[x_1 \mapsto \text{“}a\text{”}]}) = b$?
If no, output 1.
If yes, output 0.

Here is a **BTEN**-command formalizing it.

$\underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}(0, a, b, \underline{0})$
$(\mathcal{B}_{1,2}(C[FRM])\mathcal{E}$;
$\underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}(FRM(a), a, b, - - -)$
if $x_0 \approx 0$
$\underline{\qquad\qquad\qquad\qquad\qquad\qquad}(0, - - -)$
thendo $x_0 \leftarrow 1$
$\underline{\qquad\qquad\qquad\qquad\qquad\qquad}(\underline{1}, - - -) \qquad (1, a, b, - - -)$
elsedo $(x_3 \leftarrow x_2$;
$\underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}(-, a, -, b, - - -)$
$\mathcal{B}_{1,3}C[SYN]\mathcal{E}$;
$\underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}(SYN(a), a, -, b, - - -)$
$\mathcal{B}_3(x_2 \leftarrow x_0$; $C[SNF_1])\mathcal{E}$; $\underline{\qquad}(-, a, SYN(a), -, - - -)$
$\underline{\qquad\qquad\qquad\qquad\qquad\qquad}(SNF_1(a, SYN(a)), -, -, b, - - -)$
if $x_0 \approx x_3$
$\underline{\qquad\qquad\qquad\qquad\qquad\qquad}(b, -, -, b, - - -)$
thendo $x_0 \leftarrow 0$
$\underline{\qquad\qquad\qquad\qquad\qquad\qquad}(\underline{0}, - - -) \qquad (\neq b, -, -, b, - - -)$
elsedo $x_0 \leftarrow 1$))
$\underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}(\underline{1}, - - -)$

VII. The λ-calculus

This is quite a rich subject, but rather under-appreciated in many mathematical circles, apparently. It has 'ancient' connections to the foundations of both mathematics and also of computability, and more recent connections to functional programming languages and denotational semantics in computer science. This write-up is largely independent of the lengthier Computability for the Mathematical—[**CM**], within which it will also appear.

Do not misconstrue it as negative criticism, but I find some of the literature on the λ-calculus to be simultaneously quite stimulating and somewhat hard to read. Reasons for the latter might be:

(1) neuron overload caused by encyclopædiae of closely related, but subtly different, definitions (needed for deeper knowledge of the subject perhaps, or maybe a result of the experts having difficulty agreeing on which concepts are central and which peripheral—or of me reading only out-of-date literature!);

(2) authors whose styles might be affected by a formalist philosophy of mathematics ('combinatory logicians'), while I am often trying to 'picture actual existing abstract objects', in my state of Platonist original sin; and

(3) writing for readers who are already thoroughly imbued (and so, familiar with the jargon and the various 'goes without saying's) either as professional universal algebraists/model theorists or as graduate students/professionals in theoretical computer science.

We'll continue to write for an audience assumed to be fairly talented upper year undergraduates specializing in mathematics. And so we expect a typical (perhaps unexamined) Platonist attitude, and comfortable knowledge of functions, equivalence relations, abstract symbols, etc., as well as, for example, using the '=' symbol between the names of objects only when wishing to assert that they are (names for) the same object.

The text [**HS**] is one of the definite exceptions to the 'hard for me to read' comment above. It is very clear everywhere. But it requires more familiarity with logic and model theory (at least the notation) than we need below. It uses model theoretic language in doing the material we cover, and tends to be more encyclopædic, though nothing like [**Ba**]. So the following 111 or so pages will hopefully serve a useful purpose, including re-expressing things such as combinatorial completeness and the 'equivalence' between combinators and λ-calculus in an alternative, more familiar language for some of us beginners. Also it's nice not to need to be continually reminding readers

that "=" is used for identicalness of objects, with the major exception of the objects most discussed (the terms in the λ-calculus), where a different symbol must be used, because "=" has been appropriated to denote every equivalence relation in sight! At risk of boring the experts who might stumble upon this paper, it will include all but the most straightforward of details, and plenty of the latter as well (and stick largely to the extensional case).

Contents.

VII-1 The Formal System Λ .

To get us off on the correct path, this section will introduce the λ-calculus strictly syntactically as a formal system. Motivation will be left for somewhat later than is customary.

Definition of Λ . This is defined inductively below to be a set of non-empty finite strings of symbols, these strings called *terms*. The allowable symbols are

$$\lambda \;||\; \bullet \;||\;) \;||\; (\;||\; x_0 \;||\; x_1 \;||\; x_2 \;||\; x_3 \cdots$$

All but the first four are called *variables*. The inductive definition gives Λ as the smallest set of strings of symbols for which (i) and (ii) below hold:

(i) Each $x_i \in \Lambda$ (atomic terms).

(ii) If A and B are in Λ, and x is a variable, then

$$(AB) \in \Lambda \qquad \text{and} \qquad (\lambda x \bullet A) \in \Lambda \; .$$

Definition of free and bound occurrences of variables.

Notice that the definition below is exactly parallel to the corresponding one in 1$^{\underline{\text{st}}}$order logic, **[LM], Section 5.1**. That is, '$\lambda x\bullet$' here behaves the same as the quantifiers '$\forall x$' and '$\exists x$' in logic.

(i) The occurrence of x_i in the term x_i is free.

(ii) The free occurrences of a variable x in (AB) are its free occurrences in A plus its free occurrences in B.

(iii) There are no free occurrences of x in $(\lambda x \bullet A)$.

(iv) If y is a variable different than x, then the free occurrences of y in $(\lambda x \bullet A)$ are its free occurrences in A

(v) A *bound* occurrence is a non-free occurrence.

Definition of $A^{[x\mapsto C]}$, the substitution of the term C for the variable x in the term A.

In effect, the string $A^{[x\mapsto C]}$ is obtained from the string A by replacing each free occurrence of x in A by the string C. To be able to work with this mathematically, it is best to have an analytic, inductive definition, as below. This has the incidental effect of making it manifest that, as long as A and C *are* terms, such a replacement produces a string which *is* a term.

(i) If $x = x_i$, then $x_i^{[x\mapsto C]} := C$.

(ii) If $x \neq x_i$, then $x_i^{[x\mapsto C]} := x_i$.

(iii) $(AB)^{[x\mapsto C]} := (A^{[x\mapsto C]}B^{[x\mapsto C]})$.

(iv) $(\lambda x \bullet A)^{[x\mapsto C]} := (\lambda x \bullet A)$.

(v) If $y \neq x$, then $(\lambda y \bullet A)^{[x\mapsto C]} := (\lambda y \bullet A^{[x\mapsto C]})$.

But we really only bother with substitution when it is 'okay', as follows:

Definition of "$A^{[x\mapsto C]}$ is okay".

This is exactly parallel to the definition of substitutability in 1$^{\underline{\text{st}}}$order logic. You can give a perfectly tight inductive definition of this, consulting, if necessary, the proof in VI-3 that SUB is recursive. But somewhat more informally, it's just that, when treated as an occurrence in $A^{[x\mapsto C]}$, no free occurrence of a variable in C becomes a bound occurrence in any of the copies of C substituted for x.

(In CS texts, it is often convenient to force every substitution to be okay. So they have a somewhat more complicated definition than (i) to (v) above,

from which 'okayness' [Or should it be 'okayity'?] follows automatically.)

Bracket Removal Abbreviations.

From now on, we usually do the following, to produce strings which are not in Λ, but which are taken to be names for strings which are in fact terms in the λ-calculus.

(i) Omit outside brackets on a non-atomic term.

(ii) $A_1A_2\cdots A_n \ := \ (A_1A_2\cdots A_{n-1})A_n$ (inductively), i.e.

$$ABC = (AB)C \ \ ; \ \ ABCD = ((AB)C)D \ \ ; \ \text{ etc.}$$

(iii) $\lambda x \bullet A_1A_2\cdots A_n \ := \ \lambda x \bullet (A_1A_2\cdots A_n)$, so, for example,

$$\lambda x \bullet AB \ \neq \ (\lambda x \bullet A)B \ .$$

(iv) For any variables y_i,

$$\lambda y_1y_2\cdots y_n \bullet A \ := \ \lambda y_1 \bullet \lambda y_2 \bullet \cdots \lambda y_n \bullet A \ := \ \lambda y_1 \bullet (\lambda y_2 \bullet (\cdots (\lambda y_n \bullet A)\cdots))$$

The second equality in (iv) is the only possibility, so there's nothing to memorize there. The first equality isn't a bracket removal. Except for it (and so in the basic λ-calculus without abbreviations), it is evident that there is no need at all for the symbol '$\bullet$'. After all, except for some software engineers, who happily invent jargon very useful to them (but which confuses syntax with semantics), no logician sees the need to use $\forall x \bullet F$ in place of $\forall x F$. But the abbreviation given by the first equality in (iv) does force its use, since, for example,

$$\lambda xy \bullet x \ \neq \ \lambda x \bullet yx \ .$$

If we had dispensed with '$\bullet$' right from the beginning, the left-hand side in this display would be $\lambda x \lambda y x$, and would not be confused with its right-hand side, namely λxyx .

There is another statement needed about abbreviations, which 'goes without saying' in most of the sources I've read. But I'll say it:

(v) If A is a term, and S, T are (possibly empty) strings of symbols such that SAT is a term, and if R is an abbreviated string for A as given by some of (ii) to (iv) just above, then $SRT := SAT$; that is, the abbreviations in (ii) to (iv) apply to *subterms*, not just to entire terms. But (i) doesn't of

course; one must restore the outside brackets on a term when it is used as a subterm of a larger term.

We should comment about the definition of the occurrences of subterms. The implicit definition above is perfectly good to begin with: a subterm occurrence is simply a connected substring inside a term, as long as the substring is also a term when considered on its own.

However, there is a much more useful inductive definition just below. Some very dry stuff, as in Appendix B of **[LM]**, shows that the two definitions agree. The inductive definition is:

(0) Every term is a subterm occurrence in itself.

(i) The atomic term x_i has no proper subterm occurrences.

(ii) The proper subterm occurrences in (AB) are all the subterm occurrences in A together with all the subterm occurrences in B .

(iii) The proper subterm occurrences in $(\lambda x \bullet A)$ are all the subterm occurrences in A .

Finally we come to an *interesting* definition! But why this is so will remain a secret for a few more pages.

Definition of the equivalence relation $\approx$.

This is defined to be the smallest equivalence relation on Λ for which the following four conditions hold for all A, B, A', B', x and y:

(α) : $\lambda x \bullet A \ \approx \ \lambda y \bullet A^{[x \to y]}$ if $A^{[x \to y]}$ is okay and y has no free occurrence in A.

(β) : $(\lambda x \bullet A)B \ \approx \ A^{[x \to B]}$ if $A^{[x \to B]}$ is okay.

(η) : $\lambda x \bullet (Ax) \ \approx \ A$ if A has no free occurrence of x .

(Congruence Condition) :

$$[A \approx A' \quad \text{and} \quad B \approx B'] \implies [AB \ \approx \ A'B' \quad \text{and} \quad \lambda x \bullet A \ \approx \ \lambda x \bullet A'] \ .$$

Exercise. Show that

$$\lambda x \bullet A \ \approx \ \lambda x \bullet A' \implies A \approx A' \ .$$

Remarks. The names $(\alpha), (\beta)$ and (η) have no significance as far as I know, but are traditional. The brackets on the left-hand side of (η) are not

necessary. Most authors in this subject seem to use $A \equiv B$ for our $A = B$, and then use $A = B$ for our $A \approx B$. I strongly prefer to reserve '=' to mean 'is the same string as', or more generally, 'is the same thing as', for things from Plato's world. Think about the philosophy, not the notation, but do get the notation clear.

And most importantly, this definition more specifically says that *two terms are related by '$\approx$' if and only if* there is a finite sequence of terms, starting with one and ending with the other, such that each step in the sequence (that is, successive terms) comes from $(\alpha), (\beta)$ or (η) being applied to a subterm of some term, where those three 'rules' may be applied in either direction.

The last remark contains a principle, which again seems to go without saying in most expositions of the λ-calculus:

Replacement Theorem VII-1.1. *If A, B are terms, and S, T are (possibly empty) strings of symbols such that SAT is a term, then*
(i) SBT is also a term;
(ii) if $A \approx B$ then $SAT \approx SBT$.

This is not completely trivial, but is very easy by induction on the term SAT. See Theorem 2.1/2.1* in **[LM]** for an analogue.

Here are a few elementary results.

Proposition VII-1.2. *For all terms E and E_i for $1 \leq i \leq n$ such that no variable occurs in two of $E, E_1, \cdots E_n$, we have*

$$(\lambda y_1 y_2 \cdots y_n \bullet E) E_1 E_2 \cdots E_n \;=\; E^{[y_1 \to E_1][y_2 \to E_2] \cdots [y_n \to E_n]} \ .$$

Proof. Proceed by induction on n, using (β) liberally. (The condition on non-common variable occurrences is stronger than really needed.)

Proposition VII-1.3. *If A and B have no free occurrences of x, and if $Ax \approx Bx$, then $A \approx B$.*

Proof. Using the 'rule' (η) for the first and last steps, and the replacement theorem (or one of the congruence conditions) for the middle step,

$$A \;\approx\; \lambda x \bullet Ax \;\approx\; \lambda x \bullet Bx \;\approx\; B \ .$$

Exercise. Show that dropping the assumption (η), but assuming instead the statement of **VII-1.3** , the resulting equivalence relation agrees with $\approx$.

Now we want to consider the process of moving from a left-hand side for one of $(\alpha), (\beta)$ or (η) to the corresponding right-hand side, but applied to a subterm, often proper. (However, for rule (α), the distinction between left and right is irrelevant.) Such a step is called a reduction.

If one such step gets from term C to term D, say that C reduces to D.

Now, for terms A and B, say that $A \overline{\succeq} B$ if and only if there is a finite sequence, starting with A and ending with B, such that each term in the sequence (except the last) reduces to the next term in the sequence.

Of course, $A \overline{\succeq} B$ implies that $A \approx B$ but the converse is false.

To show that not all terms are equivalent in the $\approx$–sense, one seems to need a rather non-trivial result, namely the following theorem.

Theorem VII-1.4.(Church-Rosser) *For all terms A, B and C, if $A \overline{\succeq} B$ and $A \overline{\succeq} C$, then there is a term D such that $B \overline{\succeq} D$ and $C \overline{\succeq} D$.*

We shall postpone the proof (maybe forever); readable ones are given in [**Kl**], and in [**HS**], Appendix 1.

Say that a term B is normal or in normal form if and only if no sequence of reductions starting from B has any step which is an application of (β) or (η) (possibly to a proper subterm). An equivalent statement is that B contains no subterm of the form of the left-hand sides of (β) or (η), i.e.

$(\lambda x \bullet A)D$,

or

$\lambda x \bullet (Ax)$ if A has no free occurrence of x.

Note that we don't require "$A^{[x \to D]}$ is okay" in the first one. Effectively, we're saying that neither reductions (β) nor (η) can ever be applied to the result of making changes of bound variables in B.

The claims just above and below do constitute little propositions, needed in a few places below, particularly establishing the following important corollary of the Church-Rosser theorem :

For all A, if $A \overline{\succeq} B$ and $A \overline{\succeq} C$ where B and C are both normal, then B and C can be obtained from each other by applications of rule (α), that is, by a change of bound variables.

So a given term has at most one normal form, up to renaming bound variables.

In particular, no two individual variables, regarded as terms, are related by $\approx$, so there are many distinct equivalence classes. As terms, variables are normal, and are not related by '$\approx$' to any other normal term, since there aren't any bound variables to rename ! But also, of course, you can

find infinitely many closed terms (ones without any free variables) which are normal, no two of which are related by '$\approx$'.

Note also that *if B is normal and $B \overline{\succeq} C$, then $C \overline{\succeq} B$.*
But a term B with this property is not necessarily normal, as the example just below shows.

VII-2 Examples and Calculations in Λ .

First, here is an example of a term which has no normal form:

$$(\lambda x \bullet xx)(\lambda x \bullet xx) \ .$$

Note that reduction (β) applies to it, but it just reproduces itself. To prove rigorously that this has no normal form, one first argues that, in a sequence of single applications of the three types of reduction, starting with the term above, every term has the form

$$(\lambda y \bullet yy)(\lambda z \bullet zz) \ ,$$

for some variables y and z. Then use the fact (not proved here) that, if A has a normal form, then there is a finite sequence of reductions, starting from A and ending with the normal form.

Though trying to avoid for the moment any motivational remarks which might take away from the mechanical formalistic attitude towards the elements of Λ (not Λ itself) which I am temporarily attempting to cultivate, it is impossible to resist remarking that the existence of terms with no normal form will later be seen to be an analogue of the existence of pairs, (algorithm, input), which do an infinite loop!

Another instructive example is, where x and y are different variables,

$$(\lambda x \bullet y)((\lambda x \bullet xx)(\lambda x \bullet xx)) \ .$$

For this one, there is an infinite sequence of reductions, leading nowhere, so to speak. Just keep working inside the brackets ending at the far right over-and-over, as with the previous example. On the other hand, applying rule (β) once to the leftmost λ, we just get y, the normal form. So not every sequence of reductions leads to the normal form, despite one existing. This example also illustrates the fact, refining the one mentioned above, that by always reducing with respect to the leftmost λ for which a (β) or (η)-reduction

exists (possibly after an (α)-'reduction' is applied to change bound variables, and doing the (β)-reduction when there is a choice between (β)and (η) for the leftmost λ), we get an algorithm which produces the normal form, if one exists for the start-term. It turns out that having a normal form is undecidable, though it is semi-decidable as the 'leftmost algorithm' just above shows.

Now comes a long list of specific elements of Λ . Actually they are mostly only well defined up to reductions using only (α), that is, up to change of bound variables. It is important only that they be well defined as equivalence classes under $\approx$. We give them as *closed* terms, that is, terms with no free variables, but leave somewhat vague which particular bound variables are to be used. Also we write them as normal forms, all except $\underline{Y}$. For example,

Definitions of $\underline{T}$ and $\underline{F}$.

$$\underline{T} \;:=\; \lambda xy \bullet x \;:=\; (\lambda x \bullet (\lambda y \bullet x)) \quad ; \quad \underline{F} \;:=\; \lambda xy \bullet y \;:=\; (\lambda x \bullet (\lambda y \bullet y)) \;,$$

where x and y are a pair of distinct variables. The second equality each time is just to remind you of the abbreviations.

Since terms obtained by altering x and y are evidently equivalent to those above, using (α), the classes of $\underline{T}$ and $\underline{F}$ under $\approx$ are independent of choice of x and y. Since all the propositions below concerning $\underline{T}$ and $\underline{F}$ are 'equations' using '$\approx$', not '=', the choices above for x and y are irrelevant.

We shall continue for this one section to work in an unmotivated way, just grinding away in the formal system, by which I mean roughly the processing of strings by making reductions, as defined a few paragraphs above. But the notation for some of the elements defined below is quite suggestive as to what is going on. As Penrose [**Pe**], p. XXXX has said, some of this, as we get towards the end of this section, is "magical"... and ... "astonishing"! See also the exercise in the next subsection.

VII-2.1 *For all terms B and C, we have*
(a) $\underline{T}BC \approx B$; and (b) $\underline{F}BC \approx C$.

Proofs. For the latter, note that

$$\underline{F}B = (\lambda x \bullet (\lambda y \bullet y))B \approx (\lambda y \bullet y)^{[x \mapsto B]} = \lambda y \bullet y \;.$$

Thus

$$\underline{F}BC = (\underline{F}B)C \approx (\lambda y \bullet y)C \approx y^{[y \mapsto C]} = C \;.$$

For **VII-2.1**(a), choose some variable z not occurring in B, and also different from x. Then

$$\underline{T}B = (\lambda x \bullet (\lambda y \bullet x))B \approx (\lambda x \bullet (\lambda z \bullet x))B \approx (\lambda z \bullet x)^{[x \mapsto B]} = \lambda z \bullet B \ .$$

Thus

$$\underline{T}BC = (\underline{T}B)C \approx (\lambda z \bullet B)C \approx B^{[z \mapsto C]} = B \ .$$

Definition of $\underline{\neg}$. Now define

$$\underline{\neg} \ := \ \lambda z \bullet z \ \underline{F} \ \underline{T} \ = \ \lambda z \bullet ((z \ \underline{F}) \ \underline{T}) \ \neq \ (\lambda z \bullet z) \ \underline{F} \ \underline{T} \ ,$$

for some variable z. Once again, using rule (α), this is independent, up to $\approx$, of the choice of z.

VII-2.2 *We have* $\underline{\neg} \ \underline{T} \approx \underline{F}$ *and* $\underline{\neg} \ \underline{F} \approx \underline{T}$.

Proof. Using **VII-2.1**(a) for the last step,

$$\underline{\neg} \ \underline{T} = (\lambda z \bullet z \ \underline{F} \ \underline{T})\underline{T} \approx (z \ \underline{F} \ \underline{T})^{[z \mapsto \underline{T}]} = \underline{T} \ \underline{F} \ \underline{T} \approx \underline{F} \ .$$

Using **VII-2.1**(b) for the last step,

$$\underline{\neg} \ \underline{F} = (\lambda z \bullet z \ \underline{F} \ \underline{T})\underline{F} \approx (z \ \underline{F} \ \underline{T})^{[z \mapsto \underline{F}]} = \underline{F} \ \underline{F} \ \underline{T} \approx \underline{T} \ .$$

Definitions of $\underline{\wedge}$ and $\underline{\vee}$. Let

$$\underline{\wedge} \ := \ \lambda xy \bullet (x \ y \ \underline{F}) \ \text{ and } \ \underline{\vee} \ := \ \lambda xy \bullet (x \ \underline{T} \ y) \ \ ,$$

for a pair of distinct variables x and y.

VII-2.3 *We have*

$$\underline{\wedge} \ \underline{T} \ \underline{T} \approx \underline{T} \ \ ; \ \ \underline{\wedge} \ \underline{T} \ \underline{F} \approx \underline{\wedge} \ \underline{F} \ \underline{T} \approx \underline{\wedge} \ \underline{F} \ \underline{F} \approx \underline{F} \ ,$$

and

$$\underline{\vee} \ \underline{T} \ \underline{T} \approx \underline{\vee} \ \underline{T} \ \underline{F} \approx \underline{\vee} \ \underline{F} \ \underline{T} \approx \underline{T} \ \ ; \ \ \underline{\vee} \ \underline{F} \ \underline{F} \approx \underline{F} \ .$$

Proof. This is a good exercise for the reader to start becoming a λ-phile.

Definitions of $\underline{1st}$, $\underline{rst}$ and $[A,B]$.

$$\underline{1st} \;:=\; \lambda x \bullet x\ \underline{T} \;\;;\;\; \underline{rst} \;:=\; \lambda x \bullet x\ \underline{F} \;\;;\;\; [A,B] \;:=\; \lambda x \bullet xAB \;\;,$$

where x is any variable for the first two definitions, but, for the latter definition, must not occur in A or B, which are any terms.

Note that $[A,B]$ is very different than (AB)

VII-2.4 *We have*

$$\underline{1st}[A,B] \approx A \quad \text{and} \quad \underline{rst}[A,B] \approx B \;.$$

Proof. Choosing y not occurring in A or B, and different than x,

$$\underline{1st}[A,B] = (\lambda x \bullet x\ \underline{T})(\lambda y \bullet yAB) \approx (x\ \underline{T})^{[x \mapsto (\lambda y \bullet yAB)]}$$

$$= (\lambda y \bullet yAB)\underline{T} \approx \underline{T}AB \approx A \;,$$

using **VII-2.1**(a). The other one is exactly parallel.

Definitions of $\underline{ith}$ and $[A_1, A_2, \cdots, A_n]$.

Inductively, for $n \geq 3$, define

$$[A_1, A_2, \cdots, A_n] \;:=\; [A_1, [A_2, \cdots, A_n]] \;,$$

so $[A,B,C] = [A,[B,C]]$ and $[A,B,C,D] = [A,[B,[C,D]]]$, etc.

Thus, quoting **VII-2.4**,

$$\underline{1st}[A_1, A_2, \cdots, A_n] \approx A_1 \quad \text{and} \quad \underline{rst}[A_1, A_2, \cdots, A_n] \approx [A_2, A_3, \cdots, A_n] \;.$$

Here $[A]$ means A, when $n = 2$. We wish to have, for $1 \leq i < n$,

$$\underline{(i+1)th}[A_1, A_2, \cdots, A_n] \approx A_{i+1} \approx \underline{ith}[A_2, \cdots, A_n] \approx \underline{ith}(\underline{rst}[A_1, A_2, \cdots, A_n]) \;.$$

So it would be desirable to define $\underline{ith}$ inductively so that

$$\underline{(i+1)th}\ E \;\approx\; \underline{ith}\ (\underline{rst}\ E)$$

for all terms E. But we can't just drop those brackets and 'cancel' the E for this !! However, define

$$\underline{(i+1)th} \;:=\; \underline{B}\ \underline{ith}\ \underline{rst} \;,$$

where

$$\underline{S} \;:=\; \lambda xyz \bullet (xz)(yz) \quad \text{and} \quad \underline{B} \;:=\; \underline{S}\,(\underline{T}\,\underline{S})\,\underline{T}\;.$$

(Of course $\underline{1th}$ is the same as $\underline{1st}$, and maybe we should have alternative names $\underline{2nd}$ and $\underline{3rd}$ for $\underline{2th}$ and $\underline{3th}$!) This is all we need, by the second 'equation' below:

VII-2.5 *We have, for all terms A, B and C,*

$$\underline{S}ABC \approx (AC)(BC) \quad \text{and} \quad \underline{B}ABC \approx A(BC)\;.$$

Proof. The first equation is a mechanical exercise, using (β) three times, after writing $\underline{S}$ using bound variables that don't occur in A, B or C. Then the second one is a good practice in being careful with brackets, as follows:

$$\underline{B}ABC = \underline{S}\,(\underline{T}\,\underline{S})\,\underline{T}ABC = (\underline{S}\,(\underline{T}\,\underline{S})\,\underline{T}A)BC \approx (\underline{T}\,\underline{S})A(\underline{T}A)BC$$

$$= (\underline{T}\,\underline{S}\,A)(\underline{T}A)BC \approx \underline{S}(\underline{T}A)BC \approx (\underline{T}A)C(BC) = (\underline{T}AC)(BC) \approx A(BC)\;.$$

We have twice used both the first part and **VII-2.1**(a).

Don't try to guess what's behind the notation $\underline{S}$ and $\underline{B}$—they are just underlined versions of the traditional notations for these 'operators'. So I used them, despite continuing to use B (not underlined) for one of the 'general' terms in stating results. For those who have already read something about combinatory algebra, later we shall also denote $\underline{T}$ alternatively as $\underline{K}$, so that $\underline{S}$ and $\underline{K}$ are the usual notation (underlined) for the usual two generators for the *combinators*. The proof above is a foretaste of *combinatory logic* from three subsections ahead. But we could have just set

$$\underline{B} \;:=\; \lambda xyz \bullet x(yz)\;,$$

for our purposes here, and proved that $\underline{B}ABC \approx A(BC)$ directly.

Definitions of $\underline{I}$ and A^n . Define, for any term A,

$$A^0 \;:=\; \underline{I} \;:=\; \underline{S}\,\underline{T}\,\underline{T}\;,$$

and inductively

$$A^{n+1} \;:=\; \underline{B}\; A\; A^n \;.$$

VII-2.6 *For all terms A and B, and all natural numbers i and j, we have*

$$A^0 B = \underline{I}\; B \approx B \quad ; \quad \underline{I} \approx \lambda x \bullet x \quad ; \text{ and } \quad A^i(A^j B) \approx A^{i+j} B \;.$$

Also $A^1 \approx A$.

Proof. This is a good exercise, using induction on i for the 3rd displayed identity.

Exercise. (i) Sometimes $A^2 \not\approx AA$. For example, try $A = \underline{T}$ or $\underline{F}$.

(ii) Do the following sometimes give four distinct equivalence classes of terms :

$$A^3 \;\; ; \;\; AA^2 \;\; ; \;\; AAA = (AA)A \;\; ; \;\; A(AA) \approx A^2 A \;?$$

Definitions of $\underline{s}$, $\underline{isz}$ and $\overline{n}$. Define, for natural numbers n,

$$\overline{n} \;:= \lambda uv \bullet u^n v \quad ; \quad \underline{s} \;:= \lambda xyz \bullet (xy)(yz) \quad ; \quad \underline{isz} \;:= \lambda x \bullet x(\lambda y \bullet \underline{F})\underline{T} \;.$$

Here, u and v are variables, different from each other, as are x, y and z.

VII-2.7 *We have, for all natural numbers n,*

$$\underline{s}\; \overline{n} \approx \; \overline{n+1} \quad ; \quad \underline{isz}\; \overline{0} \approx \underline{T} \quad ; \quad \underline{isz}\; \overline{n} \approx \underline{F} \text{ if } n > 0 \;.$$

Proof. For any n, using four distinct variables,

$$\underline{isz}\; \overline{n} = (\lambda x \bullet x(\lambda y \bullet \underline{F})\underline{T})(\lambda uv \bullet u^n v) \approx (\lambda uv \bullet u^n v)(\lambda y \bullet \underline{F})\underline{T}$$

$$\approx (\lambda v \bullet (\lambda y \bullet \underline{F})^n v)\underline{T} \approx (\lambda y \bullet \underline{F})^n \underline{T} \qquad (*)$$

So when $n = 0$, we get

$$\underline{isz}\; \overline{0} \approx (\lambda y \bullet \underline{F})^0 \underline{T} \approx \underline{T} \;,$$

since $A^0 B \approx B$ quite generally.

Note that $(\lambda y \bullet \underline{F})B \approx \underline{F}$ for any B, merely because $\underline{F}$ is a closed term. So when $n > 0$, by $(*)$ and **VII-2.6** with $i = 1$ and $j = n - 1$,

$$\underline{isz}\; \overline{n} \approx (\lambda y \bullet \underline{F})((\lambda y \bullet \underline{F})^{n-1}\underline{T}) \approx \underline{F} \;.$$

The latter '$\approx$' is from the remark just before the display.

Finally, for any closed term B and many others, the definition of $\underline{s}$ and rule (β) immediately give

$$\underline{s}\ B \approx \lambda yz \bullet By(yz)$$

So

$$\underline{s}\ \overline{n} \approx \lambda yz \bullet (\lambda uv \bullet u^n v)y(yz) \approx \lambda yz \bullet (\lambda v \bullet y^n v)(yz)$$

$$\approx \lambda yz \bullet y^n(yz) \approx \lambda yz \bullet y^{n+1}z) \approx \ \overline{n+1}\ .$$

The penultimate step uses **VII-2.6** with $i = n$ and $j = 1$. and depends on knowing that $A^1 \approx A$.

Exercise.

(a) Define $\ \underline{+}\ :=\ \lambda uvxy \bullet (ux)(vxy)$. Show that $\underline{+}\ \overline{k}\ \overline{\ell}\ \approx\ \overline{k+\ell}$.

(m) Define $\ \underline{\times}\ :=\ \lambda uvy \bullet u(vy)$. Show that $\underline{\times}\ \overline{k}\ \overline{\ell}\ \approx\ \overline{k\ell}$.

Definitions of $\underline{pp}$ and $\underline{p}$. For distinct variables x, y, z, u and w, define

$$\underline{pp}\ := \lambda uw \bullet [\ \underline{F}\ ,\ \underline{1st}w(\underline{rst}w)(u(\underline{rst}w))\ ]\ ,$$

and

$$\underline{p}\ := \lambda xyz \bullet \underline{rst}(x(\underline{pp}\ y)[\ \underline{T}\ ,\ z\ ])\ .$$

VII-2.8 (Kleene) *For all natural numbers $n > 0$, we have* $\underline{p}\ \overline{n}\ \approx\ \overline{n-1}$.

Proof. First we show

$$\underline{pp}\ x\ [\ \underline{T}\ ,\ y\ ] \approx [\ \underline{F}\ ,\ y\ ] \qquad \text{and} \qquad \underline{pp}\ x\ [\ \underline{F}\ ,\ y\ ] \approx [\ \underline{F}\ ,\ xy\ ]\ .$$

Calculate :

$$\underline{pp}\ x\ [\ \underline{T}\ ,\ y\ ] \approx (\lambda uw \bullet [\ \underline{F}\ ,\ \underline{1st}w(\underline{rst}w)(u(\underline{rst}w))\ ])\ x\ [\ \underline{T}\ ,\ y\ ]$$

$$\approx (\lambda w \bullet [\ \underline{F}\ ,\ \underline{1st}w(\underline{rst}w)(x(\underline{rst}w))\ ])\ [\ \underline{T}\ ,\ y\ ]$$

$$\approx [\ \underline{F}\ ,\ \underline{1st}[\ \underline{T}\ ,\ y\ ](\underline{rst}[\ \underline{T}\ ,\ y\ ])(x(\underline{rst}[\ \underline{T}\ ,\ y\ ]))\ ]$$

$$\approx [\ \underline{F}\ ,\ \underline{T}\ y(xy)\ ] \approx [\ \underline{F}\ ,\ y\ ]\ .$$

The last step uses **VII-2.1**(a).

Skipping the almost identical middle steps in the other one, we get

$$\underline{pp}\ x\ [\ \underline{F}\ ,\ y\ ] \approx \cdots \approx [\ \underline{F}\ ,\ \underline{F}\ y(xy)\ ] \approx [\ \underline{F}\ ,\ xy\ ]\ .$$

Next we deduce

$$(\underline{pp}\ x)^n\ [\ \underline{F}\ ,\ y\ ] \approx [\ \underline{F}\ ,\ x^n y\ ] \qquad \text{and} \qquad (\underline{pp}\ x)^n\ [\ \underline{T}\ ,\ y\ ] \approx [\ \underline{F}\ ,\ x^{n-1}y\ ]\ ,$$

the latter only for $n > 0$. The left-hand identity is proved by induction on n, the right-hand one deduced from it, using **VII-2.6** with $i = n-1$ and $j = 1$ twice and once, respectively.

For the left-hand identity, when $n = 0$, this is trivial, in view of the fact that $A^0 B \approx B$. When $n = 1$, this is just the right-hand identity of the two proved above, in view of the fact that $A^1 \approx A$. Inductively, with $n \geq 2$, we get

$$(\underline{pp}\ x)^n\ [\ \underline{F}\ ,\ y\ ] \approx (\underline{pp}\ x)^{n-1}(\underline{pp}\ x\ [\ \underline{F}\ ,\ y\ ]) \approx (\underline{pp}\ x)^{n-1}[\ \underline{F}\ ,\ xy\ ]$$
$$\approx [\ \underline{F}\ ,\ x^{n-1}(xy)\ ] \approx [\ \underline{F}\ ,\ x^n y\ ]\ .$$

(Since xy isn't a variable, the penultimate step looks fishy, but actually, all these identities hold with x and y as names for arbitrary terms, not just variables.)

For the right-hand identity,

$$(\underline{pp}\ x)^n\ [\ \underline{T}\ ,\ y\ ] \approx (\underline{pp}\ x)^{n-1}(\underline{pp}\ x\ [\ \underline{T}\ ,\ y\ ]) \approx (\underline{pp}\ x)^{n-1}[\ \underline{F}\ ,\ y\ ] \approx [\ \underline{F}\ ,\ x^{n-1}y\ ]\ .$$

Now to prove Kleene's striking (see the next subsection) discovery, using the (β)-rule with the definition of $\underline{p}$, we see that, for all terms A,

$$\underline{p}\ A\ \approx\ \lambda yz \bullet \underline{rst}(A(\underline{pp}\ y)[\ \underline{T}\ ,\ z\ ])\ .$$

Thus

$$\underline{p}\,\overline{n}\ \approx\ \lambda yz{\bullet}\underline{rst}((\lambda uw{\bullet}u^n w)(\underline{pp}\,y)[\,\underline{T}\,,\,z\,])\ \approx\ \lambda yz{\bullet}\underline{rst}((\lambda w{\bullet}(\underline{pp}\,y)^n w)[\,\underline{T}\,,\,z\,])$$
$$\approx\ \lambda yz \bullet \underline{rst}((\underline{pp}\ y)^n[\ \underline{T}\ ,\ z\ ])\ \approx\ \lambda yz \bullet \underline{rst}[\ \underline{F}\ ,\ y^{n-1}z\ ])$$
$$\approx\ \lambda yz{\bullet}y^{n-1}z\ \approx\ \overline{n-1}\,.$$

Exercise. Show that $\underline{p}\ \overline{0}\ \approx \overline{0}$.

Exercise. Do we have $\underline{p}\ \underline{s}\ \approx \underline{I}$ or $\underline{s}\ \underline{p}\ \approx \underline{I}$?

Definition of $\underline{Y}$.

$$\underline{Y}\ := \lambda x \bullet (\lambda y \bullet x(yy))(\lambda y \bullet x(yy)) \ .$$

Of course, x and y are different from each other.

> *It is a very powerful concept, one that in a sense has inspired many mathematical developments, such as recursion theory, indeed also some literary productions, such as Hofstadter's popular book.*
>
> Erwin Engeler **[En]**

VII-2.9 *For all terms A, there is a term B such that $AB \approx B$. In fact, the term $B = \underline{Y}\ A$ will do nicely for that.*

Proof. Using the (β)-rule twice, we see that

$$\underline{Y}\ A\ =\ (\ \lambda x \bullet (\lambda y \bullet x(yy))(\lambda y \bullet x(yy))\ )A\ \approx\ (\lambda y \bullet A(yy))(\lambda y \bullet A(yy))$$

$$\approx\ A((\lambda y \bullet A(yy))(\lambda y \bullet A(yy)))\ \approx\ A\ (\underline{Y}\ A)\ ,$$

as required. The last step used the '$\approx$' relation between the first and third terms in the display.

It is interesting that Curry's $\underline{Y}$ apparently fails to give either $A(\underline{Y}A) \overline{\succeq} \underline{Y}A$, or $\underline{Y}A \overline{\succeq} A(\underline{Y}A)$. It can be useful to replace it by Turing's

$$\underline{Z}\ :=\ (\lambda xy \bullet y(xxy))(\lambda xy \bullet y(xxy))\ ,$$

another so-called fixed point operator for which at least one does have $\underline{Z}A \overline{\succeq} A(\underline{Z}A)$ for all A in Λ.

Recall that a *closed* term is one with no free variable occurrences.

VII-2.10 *Let x and y be distinct variables, and let A be a term with no free variables other than possibly x and y. Let $F := \underline{Y}(\lambda yx \bullet A)$. Then, for any closed term B, we have*

$$FB\ \approx\ A^{[y \to F]\,[x \to B]}\ .$$

The 'okayness' of the substitutions below follows easily from the restrictions on A and B, which are stronger than strictly needed, but are satisfied in all the applications.

Proof. Using the basic property of $\underline{Y}$ from **VII-2.9** for the first step,

$$FB \;\approx\; (\lambda yx \bullet A)FB \;\approx\; (\lambda x \bullet A)^{[y\mapsto F]}B \;\approx\; (\lambda x \bullet A^{[y\mapsto F]})B \;\approx\; A^{[y\mapsto F][x\mapsto B]} \;.$$

VII-2.11 *For all terms G and H, there is a term F such that*

$$F\,\overline{0} \;\approx\; G \quad \textit{and} \quad F\,\overline{n+1} \;\approx\; H[\,F\overline{n}\;,\;\overline{n}\,] \;\; \textit{for all natural numbers } n \;.$$

(My goodness, this looks a lot like primitive recursion!)

Proof. Let

$$F := \underline{Y}(\lambda yx \bullet A) \;,$$

as in the previous result, where x and y are distinct variables which do not occur in G or H, and where

$$A \;:=\; (\underline{isz}\;x)\;G\;(H[\;y(\underline{p}x)\;,\;\underline{p}x\;]) \;.$$

Then, for any term B, using **VII-2.10** for the first step,

$$FB \;\approx\; A^{[y\mapsto F][x\mapsto B]} \;\approx\; (\underline{isz}\;B)\;G\;(H[\;F(\underline{p}B)\;,\;\underline{p}B\;]\;) \;.$$

First take $B = \overline{0}$. But $\underline{isz}\;\overline{0} \approx \underline{T}$ by **VII-2.7**, and $\underline{T}GJ \approx G$ by **VII-2.1**(a), so this time we get $F\,\overline{0} \;\approx\; G$, as required.

Then take $B = \overline{n+1}$. But $\underline{isz}\;\overline{n+1} \approx \underline{F}$ by **VII-2.7**, and $\underline{F}GJ \approx J$ by **VII-2.1**(b), so here we get

$$F\,\overline{n+1} \;\approx\; H[\;F(\underline{p}\;\overline{n+1})\;,\;\underline{p}\;\overline{n+1}] \;\approx\; H[\;F\;\overline{n}\;,\;\overline{n}\;]\;,$$

as required, using **VII-2.8** for the second step.

VII-3 So—what's going on?

Actually, I'm not completely certain myself. Let's review the various examples just presented.

At first we saw some terms which seemed to be modelling objects in propositional logic. Here are two slightly curious aspects of this. Firstly, $\underline{T}$ and $\underline{F}$ are presumably truth values in a sense, so come from the semantic side, whereas $\underline{\neg}$, $\underline{\wedge}$ and $\underline{\vee}$ are more like syntactic objects. Having them all on the same footing does alter one's perceptions slightly, at least for non-CSers. Secondly, we're not surprised to see the latter versions of the connectives acting like functions which take truth values as input, and produce truth values as output. But everything's on a symmetric footing, so writing down a term like $\underline{F}\ \underline{\wedge}$ now seems like having truth values as functions which can take connectives as input, not a standard thing to consider. And $\underline{F}\ \underline{F}$ seems even odder, if interpreted as the truth value 'substituted into itself'!

But later we had terms $\overline{n}$ which seemed to represent numbers, not functions. However, two quick applications of the β-rule yield

$$\bar{n}AB \ \approx\ A^nB \ \approx\ A(A(A\cdots(AB))\cdots)) \ .$$

So if A were thought of as representing a function, as explained a bit below, the term $\bar{n}$ may be thought of as representing that function which maps A to its n-fold iterate A^n.

Now the $\bar{n}$, along with $\underline{s}$ and $\underline{p}$ as successor and predecessor functions, or at least terms representing them, give us the beginnings of a *numeral system* sitting inside Λ. The other part of that numeral system is $\underline{isz}$, the "Is it 0?"–predicate. Of course predicates may be regarded as functions. And the final result goes a long way towards showing that terms exist to represent all primitive recursive functions. To complete the job, we just need to amplify **VII-2.11** to functions of more than one variable, and make some observations about composition. More generally, in the subsection after next, we show that *all* (possibly partial) recursive functions are definable in the λ-calculus. It is hardly surprising that no others are, thinking about Church's thesis.

Exercise. Show that $\overline{k}\ \overline{\ell}\ \approx\ \overline{\ell^k}$. Deduce that $\overline{1}$ is alternatively interpretable as a combinator which defines the exponential function. (See the exercise before **VII-2.8** for the addition and multiplication functions.)

Show that $\overline{9}\ \overline{9}\ \overline{9}\ \overline{9}\ \approx\ \overline{n}$, where $n > 2^m$, where m is greater than the mass of the Milky Way galaxy, measured in milligrams !

It is likely clear to the reader that $[A\ ,\ B\ ]$ was supposed to be a term which represents an ordered pair. And we then produced ordered n-tuple

terms in Λ, as well as terms representing the projection functions.

So we can maybe settle on regarding each term in Λ as representing a function in some sense, and the construction (AB) as having B fed in as input to the function represented by A, producing an 'answer' which again represents a function. But there is sense of unease (for some of us) in seeing what appears to be a completely self-contained system of functions, every one of which has the exact same set of all these functions apparently as both its domain and its codomain. (A caveat here—as mentioned below, the existence of terms without normal forms perhaps is interpretable as some of these functions being partial.) That begins to be worrisome, since except for the 1-element set, the set of all functions from X to itself has strictly larger cardinality than X. But nobody said it had to be *all* functions. However it still seems a bit offputting, if not inconsistent, to have a bunch of functions, every one of which is a member of its own domain! (However, the formula for $\underline{Y}$ reflects exactly that, containing 'a function which is substituted into itself'—but $\underline{Y}$ was very useful in the last result, and that's only the beginning of its usefulness, as explained in the subsection after next. The example of a term with no normal form at the beginning of Subsection VII-2 was precisely 'a function which is substituted into itself'.) The "offputting" aspect above arises perhaps from the fact that, to re-interpret this stuff in a consistent way within axiomatic $1^{\underline{\text{st}}}$order set theory, some modification (such as abandonment) of the axiom of foundation seems to be necessary. See however Scott's constructions in Subsection VII-8 ahead.

In fact, this subject was carried on from about 1930 to 1970 in a very syntactic way, with little concern about whether there might exist mathematical models for the way Λ behaved, other than Λ itself, or better, the set of equivalence classes $\Lambda/\approx$. But one at least has the Church-Rosser theorem, showing that, after factoring out by the equivalence relation $\approx$, the whole thing doesn't reduce to the triviality of a 1-element set. Then around 1970 Dana Scott produced some interesting such models, and not just because of pure mathematical interest. His results, and later similar ones by others, are now regarded as being very fundamental in parts of computer science. See Subsections VII-7, VII-8 and VII-9 ahead.

Several things remain to be considered. It's really not the terms themselves, but rather their corresponding equivalence classes under $\approx$ which might be thought of as functions. The terms themselves are more like recipes for calculating functions. Of what does the calculation consist? Presumably

its steps are the reductions, using the three rules, but particularly (β) and (η). When is a calculation finished? What is the output? Presumably the answer is that it terminates when the normal form of the start term is reached, and that's the output. Note that the numerals $\overline{n}$ themselves are in normal form (with one exception). But what if the start term has no normal form? Aha, that's your infinite loop, which of course must rear its ugly head, if this scheme is supposed to produce some kind of general theory of algorithms. So perhaps one should consider the start term as a combination of both input data and procedure. It is a fact that a normal form will eventually be produced, if the start term actually has a normal form, by always concentrating on the leftmost occurrence of λ for which an application of (β) or (η) can do a reduction (possibly preceded by an application of (α) to change bound variables and make the substitution okay). This algorithm no doubt qualifies intuitively as one, involving discrete mechanical steps and a clear 'signal', if and when it's time to stop computing. The existence of more than one universal Turing machine, more than one **ATEN**-command for computing the universal function, etc. ... presumably correspond here to the possibility of other definite reduction procedures, besides the one above, each of which will produce the normal form of the start term (input), if the start term has one. As mentioned earlier, such reduction schemes are algorithms showing the semidecidability of the existence of a normal form. Later we'll see that this question is undecidable.

We have probably left far too late a discussion of what the rules (β) and (η) are doing. And indeed, what *is* λ itself? It is often called the *abstraction*-operator. The symbol string '$\lambda x\bullet$' is to alert the computer (human or otherwise) that a function of the variable x is about to be defined. And of course it's the *free* occurrences of x in A which give the 'formula' for the function which $\lambda x \bullet A$ is supposed to be defining.

So now the explanations of (β) and (η) are fairly clear :

The rule (β) just says that to evaluate the function above on B, i.e. $(\lambda x \bullet A)B$, you substitute B for the free occurrences of x in A (of course!).

And rule (η) just says that, if x doesn't occur freely in A, then the function defined by $\lambda x \bullet (Ax)$ is just A itself (of course—'the function obtained by evaluating A on its argument!).

[As an aside, it seems to me to be not unreasonable to ask why one shouldn't change the symbol 'λ' to a symbol '$\mapsto$'. After all, that's what we're talking about, and it has already been noted that the second basic

symbol '•' is quite unneeded, except for one of the abbreviations. So the string $(\lambda x \bullet A)$ would be replaced by $(x \mapsto A)$. Some things would initially be more readable for ordinary λ-inexperienced mathematicians. I didn't want to do that in the last section, because that would have given the game away too early. And I won't do it now, out of respect for tradition. Also we can think of 'λ' as taking place in a formal language, whereas '$\mapsto$' is a concept from the metalanguage, so maybe that distinction is a good one to maintain in the notation. Actually, λ-philes will often use a sort of 'Bourbaki-λ' in their metalanguages !]

This has become rather long-winded, but it seems a good place to preview Subsection VII-6, and then talk about the history of the subject, and its successful (as well as aborted) applications.

There is a small but infinite subset of Λ called the set of combinators (up to "$\approx$", just the set of closed terms), some of whose significance was discovered by Schonfinkel around 1920, without explicitly dealing with the λ-calculus. We shall treat it in some detail in Subsection VII-6, and hopefully explain more clearly than above about just which functions in some abstract sense are being dealt with by the subject, and what it was that Schonfinkel discovered.

Independently reinventing Schonfinkel's work a few years later, Curry attempted to base an entire foundations of mathematics on the combinators, as did Church soon after, using the λ-calculus, which he invented. But the system(s) proposed (motivated by studying very closely the processes of substitution occurring in Russell & Whitehead's Principia Mathematica) were soon discovered to be inconsistent, by Kleene and Rosser, students of Church. There will be nothing on this (largely abortive) application to foundations here (beyond the present paragraph). It is the case that, for at least 45 years after the mid-1930's when the above inconsistency was discovered (and possibly to the present day), there continued to be further attempts in the same direction by a small group, a subject known as illiative combinatory logic. [To get some feeling for this, take a look at [**Fi**], but start reading at Ch.1. Skip the Introduction, at least to begin, and also the Preface, or at least don't take too seriously the claims there, before having read the entire book and about 77 others, including papers of Peter Aczel, Solomon Fefferman and Dana Scott.] It's not totally unreasonable to imagine

a foundational system in which "Everything is a function!" might be attractive. After all, our 1st order set theory version of foundations is a system in which "Everything is a set!" (and Gödel seems to tell us that, if it *is* consistent, we can never be very sure of that fact). On the other hand, it is also hardly surprising that there might be something akin to Russell's paradox, which brought down Frege's earlier higher order foundations, but in an attempted system with the combinators. The famous *fixed point combinator* $\underline{Y}$ (see **VII-2.9**), or an analogue, played a major role in the Kleene-Rosser construction showing inconsistency.

> In current mathematics ... the notion of set is more fundamental than that of function, and the domain of a function is given before the function itself. In combinatory logic, on the other hand, the notion of function is fundamental; a set is a function whose application to an argument may sometimes be an assertion, or have some other property; its members are those arguments for which the application has that property. The function is primary; its domain, which is a set, is another function. Thus it is simpler to define a set in terms of a function than vice versa; but the idea is repugnant to many mathematicians, ...
>
> H.B. Curry **[BKK-ed]**

> What I always found disturbing about combinatory logic was what seemed to me to be a *complete lack* of conceptual continuity. There were no functions known to anyone else that had the extensive properties of the combinators *and allowed self-application*. I agree that people might wish to have such functions, but very early on the contradiction found by Kleene and Rosser showed there was trouble. What I cannot understand is why there was not more discussion of the question of *how* the notion of function that was behind the theory was to be made even mildly harmonious with the "classical" notion of function. The literature on combinatorial logic seems to me to be somehow silent on this point. Perhaps the reason was that the hope of "solving" the paradoxes remained alive for a long time—and may still be alive.
>
> D. Scott **[BKK-ed]**

And that is a good lead-in for a brief discussion of the history of the λ-calculus as a foundation for computability, indeed the very first foundation, beating out Turing machines by a hair. Around 1934, Kleene made great progress in showing many known computable functions to be definable

in the λ-calculus. A real breakthrough occurred when he saw how to do it for the predecessor function, our **VII-2.8**. It was based on Kleene's results that his supervisor, Church, first made the proposal that the intuitively computable functions be identified with the mathematically defined set of λ-*definable* functions. This is of course Church's Thesis, later also called the Church-Turing Thesis, since Turing independently proposed the Turing computable functions as the appropriate set within a few months, and gave a strong argument for this being a sensible proposal. Then he proved the two sets to be the same, as soon as he saw Church's paper. The latter paper proved the famous Church's Theorem providing a negative solution to Hilbert's Entscheidungsproblem, as had Turing also done independently. See the subsection after next for the definition of λ-definable and proof that all recursive functions are λ-definable.

Finally, I understand that McCarthy, in the late 1950's, when putting forth several fundamental ideas and proposing what have come to be known as *functional programming languages* (a bit of which is the **McSELF** procedures from earlier), was directly inspired by the λ-calculus [**Br-ed**]. This lead to his invention of LISP, the first such language (though it's not purely functional, containing, as it does, imperative features). It is said that all such languages include, in some sense, the λ-calculus, or even that they are all equivalent to the λ-calculus. As a miniature example, look at **McSELF** in [**CM**]. A main feature differentiating functional from imperative programming languages (of which **ATEN** is a miniature example) is that each program, when implemented, produces steps which (instead of altering a 'store', or a sequence of 'bins' as we called it) are stages in the calculation of a function, rather like the reduction steps in reducing a λ-term to normal form, or at least attempting to do so. Clearly we should expect there to be a theorem saying that there can be no algorithm for deciding whether a λ-term *has* a normal form. See the next subsection for a version of this theorem entirely within the λ-calculus.

Another feature differentiating the two types of languages seems to be a far greater use of the so-called recursive (self-referential) programs in the functional languages. In the case of our earlier miniature languages, we see that leaving that feature out of **McSELF** would destroy it (certainly many computable functions could not be programmed), whereas one of the main points of Subsection IV-9 was to see that any recursive command could be replaced by an ordinary **ATEN**-command.

We shall spend much time in the subsection after next with explaining in detail how to use the $\underline{Y}$-operator to easily produce terms which satisfy equations. This is a basic self-referential aspect of computing with the λ-calculus. In particular, it explains the formula which was just 'pulled out of a hat' in the proof of **VII-2.11**.

We shall begin the subsection after next with another, much simpler, technical idea, which implicitly pervades the last subsection. This is how the triple product ABC can be regarded as containing the analogue of the if-then-else-construction which is so fundamental in **McSELF**.

First we present some crisp and edifying versions within the λ-calculus of familiar material from earlier in this work.

VII-4 Non-examples and Non-calculability in Λ—undecidability.

Thinking of terms in Λ as being, in some sense, algorithms, here are a few analogues of previous results related to the non-existence of algorithms/commands/Turing machines/recursive functions. All this clearly depends on the Church-Rosser theorem, since the results below would be manifestly false if, for example, all elements of Λ had turned out to be related under "$\approx$" . Nothing later depends on this subsection.

VII-4.1 (Undecidability of $\approx$.) *There is no term $\underline{E} \in \Lambda$ such that, for all A and B in Λ,*

$$\underline{E}\ A\ B\ \approx\ \begin{cases} \underline{T} & \text{if } A \approx B\ ; \\ \underline{F} & \text{if } A \not\approx B\ . \end{cases}$$

First Proof. Suppose, for a contradiction, that $\underline{E}$ did exist, and define

$$u\ :=\ \lambda y \bullet \underline{E}(yy)\overline{0}\ \overline{1}\ \overline{0}\ .$$

By the (β)-rule, we get $uu \approx \underline{E}(uu)\overline{0}\ \overline{1}\ \overline{0}$. Now either $uu \approx \overline{0}$ or $uu \not\approx \overline{0}$.

But in the latter case, we get $uu\ \approx\ \underline{F}\ \overline{1}\ \overline{0}\ \approx\ \overline{0}$, a contradiction.

And in the former case, we get $uu\ \approx\ \underline{T}\ \overline{1}\ \overline{0}\ \approx\ \overline{1}$, contradicting uniqueness of normal form, which tells us that $\overline{1}\ \not\approx\ \overline{0}$.

VII-4.2 (Analogue of Rice's Theorem.) *If $\underline{R} \in \Lambda$ is such that*

$$\forall A \in \Lambda\ \text{ either }\ \underline{R}\ A\ \approx\ \underline{T}\ \text{ or }\ \underline{R}\ A\ \approx\ \underline{F}\ ,$$

then either $\forall A \in \Lambda\ ,\ \underline{R}\ A\ \approx\ \underline{T}$ *or* $\forall A \in \Lambda\ ,\ \underline{R}\ A\ \approx\ \underline{F}$.

Proof. For a contradiction, suppose we can choose terms B and C such that $\underline{R}B \approx \underline{T}$ and $\underline{R}C \approx \underline{F}$. Then define

$$M \;:=\; \lambda y \bullet \underline{R}\; y\; C\; B \qquad \text{and} \qquad N \;:=\; \underline{Y}\; M.$$

By **VII-2.7**, we get $N \;\approx MN$. Using the (β)-rule for the second $\approx$,

$$\underline{R}N \;\approx\; \underline{R}(MN) \;\approx\; \underline{R}(\underline{R}NCB)\ .$$

Since $\underline{R}N \approx$ either $\underline{T}$ or $\underline{F}$, we get either

$$\underline{T} \;\approx\; \underline{R}(\underline{T}CB) \;\approx\; \underline{R}\,C \;\approx\; \underline{F}\ ,$$

or

$$\underline{F} \;\approx\; \underline{R}(\underline{F}CB) \;\approx\; \underline{R}\,B \;\approx\; \underline{T}\ ,$$

both of which are rather resounding contradictions to uniqueness of normal form.

Second Proof of VII-4.1. Again assuming $\underline{E}$ exists, fix any B in Λ and define $\underline{R} := \underline{E}B$. This immediately contradicts **VII-4.2**, by choosing any C with $C \not\approx B$, and calculating $\underline{R}B$ and $\underline{R}C$.

Second Corollary to VII-4.2. (Undecidability of the existence of normal form.) *There is no $\underline{N} \in \Lambda$ such that*

$$\underline{N}\ A \;\approx\; \begin{cases} \underline{T} & \text{if } A \text{ has a normal form ;} \\ \underline{F} & \text{otherwise .} \end{cases}$$

This is immediate from knowing that some, but not all, terms do have a normal form.

Third Proof of VII-4.1. As an exercise, show that there is a ' 2-variable' analogue of Rice's Theorem : that is, prove the non-existence of a nontrivial $\underline{R_2}$ such that $\underline{R_2}AB$ always has normal form either $\underline{T}$ or $\underline{F}$. Then **VII-4.1** is immediate.

VII-5 Solving equations, and proving $\mathcal{RC} \iff \lambda$-definable.

Looking back at the first two results in Subsection VII-2, namely

$$\underline{T}\ A\ B\ \approx\ A \quad \text{and} \quad \underline{F}\ A\ B\ \approx\ B\ ,$$

we are immediately reminded of if-then-else, roughly : 'a true condition says to go to A, but if false, go to B.'

Now turn to the displayed formula defining A in the proof of **VII-2.11**, i.e.

$$A\ :=\ (\underline{isz}\ x)\ G\ (H[\ y(\underline{p}x)\ ,\ \underline{p}x\ ])\ .$$

This is a triple product, which is more-or-less saying

'if x is zero, use G, but if it is non-zero, use $H[\ y(\underline{p}x)\ ,\ \underline{p}x\ ]$' .

This perhaps begins to explain why that proof works, since we are trying to get a term F which, in a sense, reduces to G when x is zero, and to something involving H otherwise. What still needs more explanation is the use of $\underline{Y}$, and the form of the term $H[\ y(\underline{p}x)\ ,\ \underline{p}x\ ]$. That explanation in general follows below; both it and the remark above are illustrated several times in the constructions further down.

The following is an obvious extension of **VII-2.10**.

Theorem VII-5.1 *Let $z, y_1, \cdots, y_k$ be mutually distinct variables and let A be any term in which no other variables occur freely. Define F to be the closed term $\underline{Y}(\lambda z y_1 \cdots y_k \bullet A)$. Then, for all closed terms $V_1, \cdots, V_n$, we have*

$$FV_1 \cdots V_n\ \approx\ A^{[z \mapsto F][y_1 \mapsto V_1] \ldots [y_k \mapsto V_k]}\ .$$

Proof. (This is a theorem, not a proposition, because of its importance, not the difficulty of its proof, which is negligible!) Using the basic property of $\underline{Y}$ from **VII-2.9** for the first step, and essentially **VII-1.2** for the other steps,

$$\begin{aligned} FV_1 \cdots V_k\ &\approx\ (\lambda z y_1 \cdots y_k \bullet A) F V_1 \cdots V_k \\ &\approx\ (\lambda y_1 \cdots y_k \bullet A)^{[z \mapsto F]} V_1 \cdots V_k\ \approx\ (\lambda y_1 \cdots y_k \bullet A^{[z \mapsto F]}) V_1 \cdots V_k \\ &\cdots\ \approx\ \cdots\ \approx A^{[z \mapsto F][y_1 \mapsto V_1] \ldots [y_k \mapsto V_k]}\ . \end{aligned}$$

Why is this interesting? Imagine given an 'equation'

$$F\ \overline{n_1} \cdots \overline{n_k}\ \approx\ ---F------F---F---\ ,$$

where the right-hand side is something constructed using the λ-calculus, except that F is an unknown term which we wish to find, and it may occur more than once on the right-hand side. We need to construct a term F which satisfies the equation. The right-hand side will probably also have the numerals $\overline{n_1} \cdots \overline{n_k}$ appearing. Well, the above theorem tells us how to solve it. Just produce the term A by inventing "$k+1$" distinct variables not occurring bound in the right-hand side above, and use them to replace the occurrences of F and of the numerals in that right-hand side. That will produce a term, since that's what we meant by saying that the right-hand side was "constructed using the λ-calculus". Now the theorem tells us the formula for F in terms of A.

Of course the theorem is more general, in that we can use any closed terms V_i in place of numerals. But very often it's something related to a function of k-tuples of numbers where the technique is used. In the specific example of the proof of **VII-2.11**, the function f to be represented can be written in the if-then-else-form as

"$f(x)$, if $x = 0$, is g, but otherwise is $h(f(\mathrm{pred}(x)), \mathrm{pred}(x))$."

Here "pred" is the predecessor function. Also g is a number represented by the term G (which is presumably a numeral), and h is a function of two variables represented by H . This immediately motivates the formula for A in the proof of **VII-2.11** .

What we called the "basic property of $\underline{Y}$ from **VII-2.9**" says that it is a so-called **fixed point operator**. There are many other possibilities for such an operator. But the detailed form of $\underline{Y}$ is always irrelevant to these constructions in existence proofs. However, when getting right down to the details of compiling a functional programming language, those details are undoubtedly essential, and there, presumably, some $\underline{Y}$'s are better than others. We gave another one, due to Turing, soon after the introduction of Curry's $\underline{Y}$ in Subsection VII-2.

Definition of the numeral equality predicate term, $\underline{eq}$.

Here is a relatively simple example, producing a useful little term for testing equality of numerals (but only numerals—testing for $\approx$ in general is

unecidable, as we saw in the previous subsection!); that is, we require

$$\underline{eq}\ \overline{n}\ \overline{k}\ \approx\ \begin{cases} \underline{T} & \text{if } n = k\ ; \\ \underline{F} & \text{if } n \neq k\ . \end{cases}$$

An informal **McSELF**-like procedure for the actual predicate would be

$\mathsf{EQ}(n,k)$ $\Leftarrow$ if $n = 0$
then if $k = 0$
then true
else false
else if $k = 0$
then false
else $\mathsf{EQ}(n-1, k-1)$

(This is not really a **McSELF** procedure for two reasons—the equality predicate is a primitive in **McSELF**, and the values should be 1 and 0, not true and false. But it does show that we could get 'more primitive' by beginning **McSELF** only with 'equality to zero' as a unary predicate in place of the binary equality predicate.)

What the last display does is to tell us a defining equation which $\underline{eq}$ must satisfy:

$$\underline{eq}\ \overline{n}\ \overline{k}\ \approx\ (\underline{isz}\ \overline{n})((\underline{isz}\ \overline{k})\underline{T}\ \underline{F})((\underline{isz}\ \overline{k})\underline{F}(\underline{eq}(\underline{p}\ \overline{n})(\underline{p}\ \overline{k})))\ .$$

Thus we merely need to take

$$\underline{eq}\ :=\ \underline{Y}(\lambda zyx \bullet A)\ ,$$

where

$$A\ :=\ (\underline{isz}\ y)((\underline{isz}\ x)\underline{T}\ \underline{F})((\underline{isz}\ x)\underline{F}(z(\underline{p}\ y)(\underline{p}\ x)))\ .$$

[Purely coincidentally, the middle term in that triple product, namely $(\underline{isz}\ x)\underline{T}\ \underline{F}$ could be replaced by simply $\underline{isz}\ x$, since $\underline{T}\ \underline{T}\ \underline{F} \approx \underline{T}$ and $\underline{F}\ \underline{T}\ \underline{F} \approx \underline{F}$.]

λ-definability

First here is the definition. In the spirit of this work, we go directly to the case of (possibly) partial functions, rather than fooling around with totals first.

Definition. Let $D \subset \mathbf{N}^k$ and let $f : D \to \mathbf{N}$. Say that the function f *is* λ*-definable* if and only if there is a term F in Λ such that the term $F\overline{n_1}\cdots\overline{n}_k$ has a normal form for exactly those $(n_1, \cdots, n_k)$ which are in D, the domain of f, and, in this case, we have

$$F\ \overline{n_1}\cdots\overline{n_k}\ \approx\ \overline{f(n_1, \cdots, n_k)}\ .$$

We wish to prove that all (partial) recursive functions are λ-definable. This definitely appears to be a somewhat involved process. In particular, despite the sentiment expressed just before the definition, it seems most efficient to deal with the λ-definability of *total* recursive functions first. So the wording of the following theorem is meant to indicate we are ignoring all the tuples not in the domains. That is, no claim is made about the non-existence of normal forms of $H\overline{n_1}\cdots\overline{n_k}$, for $(n_1, \cdots, n_k)$ not in the domain of h. This theorem is a main step for dealing with minimization (in proving all recursive functions to be λ-definable).

Theorem VII-5.2 *If $g : D_g \to \mathbf{N}$ has a Λ-term which computes it on its domain $D_g \subset \mathbf{N}^k$, then so has h, where $h(n_1, \cdots, n_k)$ is defined to be*

$$\min\{\ell \mid \ell \geq n_1\ ;\ (m, n_2, \cdots, n_k) \in D_g \text{ for } n_1 \leq m \leq \ell \text{ and } g(\ell, n_2, \cdots, n_k) = 0\}.$$

The function h has domain

$$D_h = \{(n_1, n_2, \cdots, n_k) \mid \exists \ell \text{ with } \ell \geq n_1\ ,\ g(\ell, n_2, \cdots, n_k) = 0$$
$$\text{and } (m, n_2, \cdots, n_k) \in D_g \text{ for } n_1 \leq m \leq \ell\ \}\ .$$

In particular, if g is total and h happens to be total, λ-definability of g implies λ-definability of h .

Proof. This is another example of using the fixed point operator, i.e. the term $\underline{Y}$. Here is an informal **McSELF** procedure for h :

$$\begin{array}{lll} h(n_1, \cdots, n_k) & \Leftarrow & \text{if } g(n_1, n_2, \cdots, n_k) = 0 \\ & & \text{then } n_1 \\ & & \text{else } h(n_1 + 1, n_2, \cdots, n_k) \end{array}$$

So, if G is a Λ-term defining g on its domain, then a defining equation for a term H for h is

$$H\overline{n_1}\cdots\overline{n_k}\ \approx\ (\underline{isz}\ (G\overline{n_1}\cdots\overline{n_k}))\overline{n_1}(H(\underline{s}\ \overline{n_1})\overline{n_2}\cdots\overline{n}_k)\ .$$

Thus we merely need to take

$$H \;:=\; \underline{Y}(\lambda z y_1 \cdots y_k \bullet A) \;,$$

where

$$A \;:=\; (\underline{isz}\;(Gy_1 \cdots y_k))y_1(z(\underline{s}\;y_1)y_2 \cdots y_k) \;.$$

It's surprising how easy this is, but 'that's the power of $\underline{Y}$ ' !

But to repeat: the non-existence of a normal form for $H\overline{n_1}\cdots\overline{n_k}$ away from the domain of h is quite likely false! For the next theorem, we can actually give a counter-example to establish that the proof really only gives closure of the set of *total* λ-definable functions under composition, despite the fact that, as we'll see later, the result is true in the general case of (possibly strictly) partial functions.

Theorem VII-5.3 *The set of total λ-definable functions is closed under composition. That is, given λ-definable total functions as follows :*
g, of "a" variables ; and $h_1, \cdots, h_a$, each of "b" variables ; their composition

$$(n_1, \cdots, n_b) \;\mapsto\; g(h_1(n_1, \cdots, n_b), - - -, h_a(n_1, \cdots, n_b))$$

is also λ-definable. More generally, we can again assume the functions are partial and produce a Λ-term for the composition, using ones for the ingredients; but no claim can be made about lack of normal form away from the domain.

Example. We have that the identity function, $g : \mathbf{N} \to \mathbf{N}$, is λ-defined by $G = \lambda x \bullet x$. And $H = (\lambda x \bullet xx)(\lambda x \bullet xx)$ defines the empty function $h : \emptyset \to \mathbf{N}$. The composite $g \circ h$ is also the empty function. But the term $D_{1,1}GH$ constructed in the following proof does not λ-define it; in fact, it defines the identity function.

Proof. Let terms G and $H_1, \cdots, H_a$ represent the given functions, so that

$$G\;\overline{m_1}\cdots\overline{m_a} \;\approx\; \overline{g(m_1, \cdots, m_a)}$$

and

$$H_i\;\overline{n_1}\cdots\overline{n_b} \;\approx\; \overline{h_i(n_1, \cdots, n_b)} \;,$$

each holding for those tuples for which the right-hand side is defined.

Now there is a term $D_{a,b}$ such that

$$D_{a,b}GH_1\cdots H_aN_1\cdots N_b \;\approx\; G(H_1N_1\cdots N_b)(H_2N_1\cdots N_b)---(H_aN_1\cdots N_b) \quad (*)$$

holds for *any* terms $G, H_1, \cdots, H_a, N_1, \cdots, N_b$.

Assuming this for the moment, and returning to the $G, H_1, \cdots, H_a$ earlier, the term $D_{a,b}GH_1\cdots H_a$ will represent the composition. Just calculate :

$$\begin{aligned} D_{a,b}GH_1\cdots H_a\,\overline{n_1}\cdots\overline{n_b} \;&\approx\; G(H_1\,\overline{n_1}\cdots\overline{n_b})(H_2\,\overline{n_1}\cdots\overline{n_b})---(H_a\,\overline{n_1}\cdots\overline{n_b}) \\ &\approx\; G\;\overline{h_1(n_1,\cdots,n_b)}\;\overline{h_2(n_1,\cdots,n_b)}---\overline{h_a(n_1,\cdots,n_b)} \\ &\approx\; \overline{g(h_1(n_1,\cdots,n_b),---,h_a(n_1,\cdots,n_b))}\ . \end{aligned}$$

To prove $(*)$, we give two arguments, the first being facetious:

Firstly, it's just another one of those equations to be solved using the fixed point operator, except that the number of times the unknown term $D_{a,b}$ appears on the right-hand side of its defining equation is zero. So this is a bit of a phoney application, as we see from the second proof below. In any case, by this argument, we'll set

$$D_{a,b} \;:=\; \underline{Y}(\lambda zuy_1\cdots y_ax_1\cdots x_b \bullet A)\ ,$$

for a suitable set of "$2+a+b$" variables, where

$$A \;:=\; u(y_1x_1\cdots x_b)(y_2x_1\cdots x_b)---(y_ax_1\cdots x_b)\ .$$

The other argument is a straightforward calculation, directly defining

$$D_{a,b} \;:=\; \lambda uy_1\cdots y_ax_1\cdots x_b \bullet u(y_1x_1\cdots x_b)(y_2x_1\cdots x_b)---(y_ax_1\cdots x_b)\ .$$

This more than completes the proof.

The assertion in $(*)$ is one instance of a general phenomenon called *combinatorial completeness*, which we study in the next subsection. There will be a choice between two proofs there as well, exactly analogous to what we did above. The identities in **VII-2.5** are other examples of this phenomenon.

Theorem VII-5.4 *Any total recursive function is λ-definable.*

Proof. We use the fact that any such function can be obtained from the starters discussed below, by a sequence of compositions and minimizations which use as ingredients (and produce) only *total* functions. See [**CM**]. The starters are shown to be λ-definable below, and **VII-5.3** deals with composition. So it remains only to use **VII-5.2** to deal with minimization. Suppose that $p : \mathbf{N}^{n+1} \to \mathbf{N}$ is a total function which is λ-defined by H. And assume that for all $(k_1, \cdots, k_n)$, there is a k with $p(k, k_1, \cdots, k_n) = 0$. Define the total function $m_p : \mathbf{N}^n \to \mathbf{N}$ by

$$m_p(k_1, \cdots, k_n) \ := \ \min\{k \mid p(k, k_1, \cdots, k_n) = 0\} \ .$$

Then, if $h^{(g)}$ is the h produced from g in the statement of **VII-5.2**, it is straightforward to see that $m_p = h^{(p)} \circ (\text{zero}_n, \pi_1, \cdots, \pi_n)$, where the last tuple of "n"-variable functions consists of the zero constant function and the projections, all λ-definable. But $h^{(p)}$ is λ-definable by **VII-5.2**. So, by closure under composition, the proof is complete.

Let us now have a discussion of the λ-definability of various simple functions, including the starters referred to in this last proof.

It is a total triviality to check that the constant function, mapping all numbers to zero, is defined by the term $\lambda x \bullet \overline{0}$. We already know that the successor function is defined by $\underline{s}$.

The ith projection function is *not* λ-defined by $\underline{ith}$, because we are not bothering to convert to genuine multi-valued functions, defined on tuples, but rather treating them as successively defined using adjointness. (See the initial paragraph of the Digression in Subsection VII-7 for more explanation, if desired.) There is a λ-calculus formalism for going back-and-forth between the two viewpoints, called "currying" and "uncurrying", very handy for functional programming, I believe, but we don't need it here. The ith projection function *is* defined by $\lambda x_1 x_2 \cdots x_n \bullet x_i$, as may be easily checked.

At this point, if we wish to use the starter functions just above, the proof that *every total recursive function is λ-definable* would be completed by showing that the set of λ-definable functions is closed under primitive recursion. The latter is simply the many-variable case of the curried version of **VII-2.11**, and can be safely left to the reader as yet another (by now mechanical) exercise with $\underline{Y}$.

On the other hand, we don't need to bother with primitive recursion if we go back to our original definition of the recursive functions, and also accept the theorem that a total recursive function can be built from starters using only *total* functions at all intermediate steps (as we did in this last proof). But we do need to check that the addition and multiplication functions, and the '$\geq$'-predicate, all of two variables, are λ-definable, since they were the original starters along with the projections. These are rather basic functions which everybody should see λ-defined. So we'll now do these examples of using the fixed point operator, plus one showing how we could also have gotten a different predecessor function this way. In each case we'll write down a suitable (informal) **McSELF**-procedure for the function (copying from earlier), then use it to write down the λ-calculus equation, convert that into a formula for what we've been calling A, and the job is then done as usual with an application of the fixed-point operator $\underline{Y}$.

Here is that list of terms for basic functions mentioned just above, where we've just stolen the informal **McSELF** procedures from an earlier section.

The addition function. (See also the exercise before **VII-2.8**)

$$\begin{array}{lll} \mathsf{ADD}(m,n) & \Leftarrow & \text{if } n = 0 \\ & & \text{then } m \\ & & \text{else } \mathsf{ADD}(m+1, n-1) \end{array}$$

So the term $\underline{add}$ must satisfy

$$\underline{add}\ \overline{m}\ \overline{n}\ \approx\ (\underline{isz}\ \overline{n})\overline{m}(\underline{add}(\underline{s}\ \overline{m})(\underline{p}\ \overline{n}))\ .$$

And so we take $\underline{add}\ :=\ \underline{Y}(\lambda zxy \bullet A)$ where $A\ :=\ (\underline{isz}\ y)x(z(\underline{s}\ x)(\underline{p}\ y))$.

The multiplication function. (See also the exercise before **VII-2.8**)

$$\begin{array}{lll} \mathsf{MULT}(m,n) & \Leftarrow & \text{if } n = 0 \\ & & \text{then } 0 \\ & & \text{else } \mathsf{ADD}(m, \mathsf{MULT}(m, n-1)) \end{array}$$

So the term $\underline{mult}$ must satisfy

$$\underline{mult}\ \overline{m}\ \overline{n}\ \approx\ (\underline{isz}\ \overline{n})\overline{0}(\underline{add}\ \overline{m}(\underline{mult}\ \overline{m}(\underline{p}\ \overline{n}))\ .$$

And so we take $\underline{mult}\ :=\ \underline{Y}(\lambda zxy\bullet A)$ where $A\ :=\ (\underline{isz}\ y)\underline{0}(\underline{add}\ x(z\ x(\underline{p}\ y))$.

The 'greater than or equal to' predicate.

$$\begin{array}{lll} \mathsf{GEQ}(m,n) & \Leftarrow & \text{if } n=0 \\ & & \text{then } 1 \\ & & \text{else} \quad \text{if } m=0 \\ & & \qquad \text{then } 0 \\ & & \qquad \text{else } \mathsf{GEQ}(m-1,n-1) \end{array}$$

So the term $\underline{geq}$ must satisfy

$$\underline{geq}\ \overline{m}\ \overline{n} \ \approx\ (\underline{isz}\ \overline{n})\overline{1}((\underline{isz}\ \overline{m})\overline{0}(\underline{geq}(\underline{p}\ \overline{m})(\underline{p}\ \overline{n}))) \ .$$

And so we take

$$\underline{geq} \ :=\ \underline{Y}(\lambda zxy \bullet A)$$

where

$$A \ :=\ (\underline{isz}\ y)\underline{1}((\underline{isz}\ x)\underline{0}(z(\underline{p}\ x)(\underline{p}\ y))) \ .$$

This predicate takes values 0 and 1, rather than $\underline{F}$ and $\underline{T}$. To make that alteration will be left as an exercise.

The new, non-total, predecessor function.

We get the new predecessor function PRED (undefined at 0 and otherwise mapping n to $n-1$) by $\mathsf{PRED}(n) = \mathsf{PREPRED}(0,n)$, where

$$\mathsf{PREPRED}(m,n) \ =\ \begin{cases} n-1 & \text{if } m<n \ ; \\ err & \text{if } m \geq n \ . \end{cases}$$

The latter has **McSELF**-procedure

$$\begin{array}{lll} \mathsf{PREPRED}(m,n) & \Leftarrow & \text{if } m+1=n \\ & & \text{then } m \\ & & \text{else } \mathsf{PREPRED}(m+1,n) \end{array}$$

To distinguish them from the earlier, total, functions $\underline{p}$ and $\underline{pp}$, we'll name the terms which represent these functions as $\underline{q}$ and $\underline{qq}$. Then the term $\underline{qq}$ must satisfy

$$\underline{qq}\ \overline{m}\ \overline{n} \ \approx\ (\underline{eq}(\underline{s}\ \overline{m})\overline{n})\overline{m}(\underline{qq}(\underline{s}\ \overline{m})\ \overline{n}) \ .$$

And so we take

$$\underline{qq} \ :=\ \underline{Y}(\lambda zxy \bullet A)$$

where

$$A \;:=\; (\underline{eq}(\underline{s}\;x)y)x(z(\underline{s}\;x)\;y)\;.$$

And now for the predecessor itself :

$$\underline{q}\;\overline{n} \;\;\approx\;\; \underline{qq}\;\overline{0}\;\overline{n}\;, \quad \text{so define} \quad \underline{q} \;:=\; \lambda x \bullet \underline{qq}\;\overline{0}\;x\;.$$

These are simpler than $\underline{p}$ and $\underline{pp}$, aren't they?

Now let's get serious about partial recursive functions. The theorem works for any numeral system with a few basic properties, as may be seen by checking through the proofs above and below. So that's how it will be stated:

Theorem VII-5.5. *For any choice of numeral system, that is, of terms* $\underline{s}, \underline{p}, \underline{isz}$, *and* $\overline{n}$ *for each* $n \geq 0$, *which satisfy*

$$\underline{s}\;\overline{n} \;\approx\; \overline{n+1} \;\;;\;\; \underline{isz}\;\overline{0} \approx \underline{T} \;\;;\;\; \text{if } n > 0\,, \text{then } \underline{isz}\;\overline{n} \approx \underline{F} \text{ and } \underline{p}\;\overline{n} \;\approx\; \overline{n-1}\;,$$

every (partial) recursive function is λ*-definable.*

But we'll just prove it for the earlier Church numeral system, reviewed below. Were it not for needing a term which 'does a loop', i.e. has no normal form, in 'all the right places', we would at this point have done what was necessary to prove this theorem. For obtaining the needed result about the non-existence of normal forms, it is surprising how many technicalities (just below), and particularly what difficult syntactic theorems (see the proof of the theorem a few pages ahead), seem to be needed .

We begin with a list of new and old definitions, and then four lemmas, before completing the proof.

Confession. Just below there is a slight change from an earlier definition, and also a 'mistake'—a small fib, if you like. We come clean on this right at the end of this subsection, using \sf print. This seems the best way, and it doubles the profit, as we explain there. So most readers, if they discover one or both of these two minor anomalies, should just press on in good humour. But if that is a psychological impossibility, go to the end of the subsection to get relief.

Define inductively, for Λ-terms X and Y :

$$X^m Y \;:=\; \begin{cases} Y & \text{if } m = 0\;; \\ X^{m-1}(XY) & \text{if } m > 0\;. \end{cases}$$

Thus $X^m Y \;=\; X(X(\cdots\cdots(X(XY))\cdots)) \;=\; XC$ for $C \;=\; X^{m-1}Y$.

Recall the definition $\overline{m} \;:=\; \lambda xy \bullet x^m y$.

Note that $x^m y$, despite appearances, doesn't have the form Ay for any Λ-term A, so we cannot apply (η)-reduction to reduce this to $\lambda x \bullet x^m$. In fact, $\overline{m}$ is in normal form. (In both cases, $m = 1$ is an exception.) We have

$$\overline{m}\ A\ B \,\overline{\succeq}\, A^m B \ ,$$

as is easily checked for any terms A and B, though you *do* have to check that $(x^m y)^{[x \mapsto A][y \mapsto B]} \;=\; A^m B$, which is not entirely, though almost, a no-brainer.

Recall that $\underline{s} \;:=\; \lambda xyz \bullet (xy)(yz)$. One can easily directly show that

$\underline{s}\ \overline{m}\ \overline{\succeq}\ \overline{m+1}$, though it follows from the earlier fact that $\underline{s}\ \overline{m} \;\approx\; \overline{m+1}$,

since the right-hand side is in normal form. We have also

$$IX \;\overline{\succeq}\; X \qquad \text{and} \qquad KXY \;\overline{\succeq}\; X \ ,$$

where now $\underline{K} \;:=\; \lambda xy \bullet x$ is denoted just K .

Define $D \;:=\; \lambda xyz \bullet z(Ky)x$. Then we have

$$DAB\,\overline{0} \;\overline{\succeq}\; \overline{0}(KB)A \;\overline{\succeq}\; (KB)^0 A = A \ ,$$

and, for $i > 0$,

$$DAB\,\overline{i} \;\overline{\succeq}\; \overline{i}(KB)A \;\overline{\succeq}\; (KB)^i A = KBC \;\overline{\succeq}\; B \quad , \quad \text{with } C = (KB)^{i-1}A \ .$$

Define

$$T \;:=\; \lambda x \bullet D\overline{0}(\lambda uv \bullet u(x(\underline{s}v))u(\underline{s}v)) \ ,$$

and then define, for any Λ-terms X and Y,

$$PXY \;:=\; TX(XY)(TX)Y \ .$$

If we only needed "$\approx$" and not "$\overline{\succeq}$" below, we could take P as an actual term $\lambda xy \bullet Tx(xy)(Tx)y$. Note that P will never occur on its own, only in the form PXY .

Lemma A. *For X and Y in Λ with $XY \ \overline{\succeq}\ \overline{i}$, we have*

$$PXY \ \overline{\succeq}\ \begin{cases} Y & \text{if } i = 0 \ ; \\ PX(\underline{s}Y) & \text{if } i > 0 \ . \end{cases}$$

Also, in each case, the "$\overline{\succeq}$" may be realized by a sequence of (β)-reductions, the first of which obliterates the leftmost occurrence of λ .

Proof. Using, in the first step, the definitions of PXY and of T, and a leftmost (β)-reduction,

$$PXY \ \overline{\succeq}\ D\overline{0}(\lambda uv\bullet u(X(\underline{s}v))u(\underline{s}v))(XY)(TX)Y$$

$$\overline{\succeq}\ D\overline{0}(\lambda uv \bullet u(X(\underline{s}v))u(\underline{s}v))\overline{i}(TX)Y \quad \cdots \qquad \text{(to be continued)}$$

Now when $i = 0$ this continues

$$\overline{\succeq}\ \overline{0}(TX)Y \qquad\qquad [\text{since } \ DAB\,\overline{0} \ \overline{\succeq}\ A]$$

$$\overline{\succeq}\ (TX)^0Y \ = \ Y \ .$$

When $i > 0$ it continues

$$\overline{\succeq}\ (\lambda uv \bullet u(X(\underline{s}v))u(\underline{s}v))(TX)Y \quad [\text{since } DAB\,\overline{i} \ \overline{\succeq}\ B]$$

$$\overline{\succeq}\ TX(X(\underline{s}Y))(TX)(\underline{s}Y) \ = \ PX(\underline{s}Y) \ ,$$

as required, completing the proof.

Now suppose that $g : \mathbf{N} \to \mathbf{N}$ and $h : \mathbf{N}^{n+1} \to \mathbf{N}$ are total functions, and that G, H in Λ are such that

$$G\overline{j} \ \overline{\succeq}\ \overline{g(j)} \quad \text{and} \quad H\overline{k}_1\overline{k}_2\cdots\overline{k}_n\overline{k} \ \overline{\succeq}\ \overline{h(k_1, k_2, \cdots k_n, k)}$$

for all $j, k_1, k_2, \cdots k_n, k$ in $\mathbf{N}$. Define $f : \mathrm{dom}(f) \to \mathbf{N}$ by

$$f(k_1, k_2, \cdots k_n) \ := \ g(\min\{ \ k \mid h(k_1, k_2, \cdots k_n, k) = 0 \ \}) \ ,$$

so

$$\mathrm{dom}(f) \ = \ \{ \ (k_1, k_2, \cdots k_n) \mid \exists k \text{ with } h(k_1, k_2, \cdots, k_n, k) = 0 \ \} \ .$$

Lemma B. *Given m such that $h(k_1, k_2, \cdots k_n, \ell) \neq 0$ for $0 \leq \ell < m$, we have, for these ℓ,*

(i) $P(H\overline{k}_1\overline{k}_2 \cdots \overline{k}_n)\overline{\ell} \quad \overline{\succeq} \quad P(H\overline{k}_1\overline{k}_2 \cdots \overline{k}_n)\overline{\ell+1}$,

where each such " $\overline{\succeq}$ " may be realized by a sequence of (β)-reductions at least one of which obliterates the leftmost occurrence of λ ; and

(ii) $P(H\overline{k}_1\overline{k}_2 \cdots \overline{k}_n)\overline{0} \quad \overline{\succeq} \quad P(H\overline{k}_1\overline{k}_2 \cdots \overline{k}_n)\overline{m}$.

Proof. (ii) is clearly immediate from (i). As for the latter, with $X = H\overline{k}_1\overline{k}_2 \cdots \overline{k}_n$ and $Y = \overline{\ell}$, we have

$$XY \;=\; H\overline{k}_1\overline{k}_2 \cdots \overline{k}_n\overline{\ell} \;\overline{\succeq}\; \overline{h(k_1, k_2, \cdots k_n, \ell)} \;=\; (\text{say})\; \overline{i}\ ,$$

where $i > 0$. And so

$$\begin{aligned} P(H\overline{k}_1\overline{k}_2 \cdots \overline{k}_n)\overline{\ell} \;&\overline{\succeq}\; P(H\overline{k}_1\overline{k}_2 \cdots \overline{k}_n)(\underline{s}\overline{\ell}) \quad \text{by Lemma } \mathbf{A} \text{ (second part)} \\ &\overline{\succeq}\; P(H\overline{k}_1\overline{k}_2 \cdots \overline{k}_n)\overline{\ell+1} \quad \text{since} \quad \underline{s}\overline{\ell} \;\overline{\succeq}\; \overline{\ell+1}\ . \end{aligned}$$

Lemma C. *If $(k_1, k_2, \cdots k_n)$ is such that there is a k with $h(k_1, k_2, \cdots k_n, k) = 0$, and we define*

$$m \;:=\; \min\{\ k \mid h(k_1, k_2, \cdots k_n, k) = 0\ \}\ ,$$

then $P(H\overline{k}_1\overline{k}_2 \cdots \overline{k}_n)\overline{0} \;\overline{\succeq}\; \overline{m}$.

Proof. Using Lemma B(ii), and then using Lemma A (first part) with $X = H\overline{k}_1\overline{k}_2 \cdots \overline{k}_n$ and $Y = \overline{m}$, which can be done since

$$XY \;=\; H\overline{k}_1\overline{k}_2 \cdots \overline{k}_n\overline{m} \;\overline{\succeq}\; \overline{h(k_1, k_2, \cdots k_n, m)} \;=\; \overline{0}\ ,$$

we get

$$P(H\overline{k}_1\overline{k}_2 \cdots \overline{k}_n)\overline{0} \;\overline{\succeq}\; P(H\overline{k}_1\overline{k}_2 \cdots \overline{k}_n)\overline{m} \;\overline{\succeq}\; \overline{m}\ .$$

Main Lemma D. (a) *Let*

$$L \;:=\; \lambda x_1 \cdots x_n y \bullet P(Hx_1 \cdots x_n)y \;\text{ and }\; F \;:=\; \lambda x_1 \cdots x_n \bullet G(Lx_1 \cdots x_n \overline{0}) \;.$$

Then

$$\forall (k_1, \cdots, k_n) \in \mathrm{dom}(f) \;\text{ we have }\; F\overline{k}_1\overline{k}_2 \cdots \overline{k}_n \;\overline{\succeq}\; \overline{f(k_1, k_2, \cdots k_n)} \qquad (*)$$

(b) *For any F satisfying $(*)$, define*

$$E \;:=\; \lambda x_1 \cdots x_n \bullet P(Hx_1 \cdots x_n)\overline{0}\; I(Fx_1 \cdots x_n) \;.$$

Then (i) *the reduction $(*)$ also holds with E in place of F ; and*

(ii) *for all $(k_1, \cdots, k_n) \notin \mathrm{dom}(f)$, there is an infinite sequence of (β)-reductions, starting with $E\overline{k}_1\overline{k}_2 \cdots \overline{k}_n$, and such that, infinitely often, it is the leftmost λ which is obliterated.*

Proof. (a) We have

$$\begin{aligned}
F\overline{k}_1\overline{k}_2 \cdots \overline{k}_n \;&\overline{\succeq}\; G(L\overline{k}_1\overline{k}_2 \cdots \overline{k}_n\overline{0}) && \text{[definition of } F\text{]} \\
&\overline{\succeq}\; G(P(H\overline{k}_1\overline{k}_2 \cdots \overline{k}_n)\overline{0}) && \text{[definition of } L\text{]} \\
\overline{\succeq}\; G\overline{m} \quad &\text{[where } m = \min\{\, k \mid h(k_1, k_2, \cdots k_n, k) = 0 \,\} \text{ by Lemma } \mathbf{C}\text{]} \\
\overline{\succeq}\; \overline{g(m)} \;&=\; \overline{g(\min\{\, k \mid h(k_1, k_2, \cdots k_n, k) = 0 \,\})} \;=\; \overline{f(k_1, k_2, \cdots k_n)} \;.
\end{aligned}$$

(b)(i) We have

$$\begin{aligned}
E\overline{k}_1\overline{k}_2 \cdots \overline{k}_n \;&\overline{\succeq}\; P(H\overline{k}_1\overline{k}_2 \cdots \overline{k}_n)\overline{0}I(F\overline{k}_1\overline{k}_2 \cdots \overline{k}_n) && \text{[definition of } E\text{]} \\
&\overline{\succeq}\; \overline{m}I(F\overline{k}_1\overline{k}_2 \cdots \overline{k}_n) && \text{[}m \text{ as above, by Lemma } \mathbf{C}\text{]} \\
&\overline{\succeq}\; I^m(F\overline{k}_1\overline{k}_2 \cdots \overline{k}_n) && \text{[since } \overline{m}XY \;\overline{\succeq}\; X^mY\text{]} \\
&\overline{\succeq}\; F\overline{k}_1\overline{k}_2 \cdots \overline{k}_n && \text{[since } IX \;\overline{\succeq}\; X\text{]} \\
&\overline{\succeq}\; \overline{f(k_1, k_2, \cdots k_n)} && \text{[by } (*) \text{ for } F\text{]} \;.
\end{aligned}$$

(b)(ii) We have

$$
\begin{aligned}
E\overline{k}_1\overline{k}_2\cdots\overline{k}_n \;&\overline{\succeq}\; P(H\overline{k}_1\overline{k}_2\cdots\overline{k}_n)\;\overline{0}\;I(F\overline{k}_1\overline{k}_2\cdots\overline{k}_n) && \text{[definition of } E\text{]}\\
&\overline{\succeq}\; P(H\overline{k}_1\overline{k}_2\cdots\overline{k}_n)\;\overline{1}\;I(F\overline{k}_1\overline{k}_2\cdots\overline{k}_n) && \text{[repeatedly applying}\\
&\overline{\succeq}\; P(H\overline{k}_1\overline{k}_2\cdots\overline{k}_n)\;\overline{2}\;I(F\overline{k}_1\overline{k}_2\cdots\overline{k}_n) && \text{Lemma } \mathbf{B}\text{(i), and}\\
&\overline{\succeq}\; P(H\overline{k}_1\overline{k}_2\cdots\overline{k}_n)\;\overline{3}\;I(F\overline{k}_1\overline{k}_2\cdots\overline{k}_n) && \text{noting that each step}\\
&\overline{\succeq}\; P(H\overline{k}_1\overline{k}_2\cdots\overline{k}_n)\;\overline{4}\;I(F\overline{k}_1\overline{k}_2\cdots\overline{k}_n) && \text{here is doable with}\\
&\overline{\succeq}\; P(H\overline{k}_1\overline{k}_2\cdots\overline{k}_n)\;\overline{5}\;I(F\overline{k}_1\overline{k}_2\cdots\overline{k}_n) && \text{at least one leftmost}\\
&\overline{\succeq}\; P(H\overline{k}_1\overline{k}_2\cdots\overline{k}_n)\;\overline{6}\;I(F\overline{k}_1\overline{k}_2\cdots\overline{k}_n) && (\beta)-\text{reduction, giving}\\
&\overline{\succeq}\; P(H\overline{k}_1\overline{k}_2\cdots\overline{k}_n)\;\overline{7}\;I(F\overline{k}_1\overline{k}_2\cdots\overline{k}_n) && \text{infinitely many,}\\
&\overline{\succeq}\; P(H\overline{k}_1\overline{k}_2\cdots\overline{k}_n)\;\overline{8}\;I(F\overline{k}_1\overline{k}_2\cdots\overline{k}_n) \quad \bullet\bullet\bullet && \text{as required.]}
\end{aligned}
$$

Proof of Theorem VII-5.5—*Any (partial) recursive function is λ-definable.* Let f be such a function, and use Kleene's theorem (see [**CM**], Section IV) to write

$$f(k_1, k_2, \cdots k_n) \;=\; g(\min\{\; k \mid h(k_1, k_2, \cdots k_n, k) = 0 \;\})\;,$$

for primitive recursive functions g and h. These two are, in particular, *total* recursive functions. So we can, by **VII-5.4**, find G and H in Λ such that

$$G\overline{j} \;\approx\; \overline{g(j)} \qquad \text{and} \qquad H\overline{k}_1\overline{k}_2\cdots\overline{k}_n\overline{k} \;\approx\; \overline{h(k_1, k_2, \cdots k_n, k)}$$

for all $j, k_1, k_2, \cdots k_n, k$ in $\mathbf{N}$. But since the right-hand sides are already in normal form, by the normal form theorem we can change "$\approx$" to "$\overline{\succeq}$" in both places above. But now, using parts of Lemma **D**, first get an F as in part (a), and then use it to get an E as in part (b). Part (b)(i) gives the "$\approx$" equation required for $(k_1, \cdots, k_n) \in \mathrm{dom}(f)$ (and more, since it even gives a specific way to reduce $E\overline{k}_1\overline{k}_2\cdots\overline{k}_n$ to $\overline{f(k_1, k_2, \cdots k_n)}$, dependent on having similar reductions for G and H—see the exercises below as well). For $(k_1, \cdots, k_n) \notin \mathrm{dom}(f)$, part (b)(ii) gives an infinite reduction sequence starting with $E\overline{k}_1\overline{k}_2\cdots\overline{k}_n$, and such that, infinitely often, it is the leftmost λ which is obliterated. Thus it is a "quasi-leftmost reduction" in the sense of [**Kl**], because of leftmost (β)-reductions occurring arbitrarily far out. Thus,

by Cor 5.13, p.293 of that thesis, $E\overline{k}_1\overline{k}_2\cdots\overline{k}_n$ has no normal form (in the *extensional* λ-calculus), as required.

It seems interesting how much is involved in this proof. I've not seen any essentially different proof; in fact I've not seen any other detailed proof at all in the extensional case. This one is essentially that in [**HS**], except they deal with the *intensional* λ-calculus. And they are acknowledged experts. One can well imagine manœuvering to avoid Kleene's theorem. But Klop's quasi-leftmost normalization theorem seems to be essential, and its proof apparently dates from only around 1980. It is stated as a problem by Hindley [**Hi**] in 1978. Furthermore, it seems always to be the extensional λ-calculus with which the CSers work in hard-core denotational semantics. I would assume that a definition of "computable" phrased in terms of the extensional λ-calculus was one which was long accepted before 1980. This may be a situation somewhat analogous to that with the Littlewood-Richardson rule from algebraic combinatorics/representation theory. Those two only stated the rule in the 1930's as an empirically observed phenomenon. A proof or proofs with large gaps appeared long before 1970, but only around then did acceptable proofs begin to appear. A joke I heard from Ian Macdonald has astronauts landing on the moon and (fortunately) returning, by use of technology dependent on the Littlewood-Richardson rule, apparently well before a genuine proof was known!

In any case, let me own up again—we have, in this write-up, depended on three basic, difficult syntactic properties of the extensional λ-calculus (unproved here) :

the Church-Rosser theorem and its consequence that normal forms are unique;

the normalization theorem, that when M has a normal form N, the sequence of leftmost reductions starting with M terminates with N ;

and Klop's theorem above, implying that, if an infinite quasi-leftmost reduction starting from M exists, then M has no normal form (which theorem also says that we can weaken the word "leftmost" to "quasi-leftmost" in the preceding statement about those M's which *do* have normal forms).

Coming clean. The minor change, from earlier definitions in VII-2, made just after the "Confession", is that here in VII-4 we defined X^mY a bit differently. The new definition agrees with the old up to "$\approx$", so everything stated earlier still

holds using the new definition. Since they are defined using $x^m y$, the numerals $\overline{m}$ have now been changed slightly, so there are actually quite a few earlier results to which that comment applies. It just seemed to the author that in VII-2 it would avoid fuss if X^m actually had a meaning on its own as a term. From the "Confession" onwards, that definition should be ignored—X^m never occurs without being followed by a term Y.

The mistake referred to in the "Confession" is that, in the extensional λ-calculus, though all the other numerals are, the numeral $\overline{1}$ is NOT in normal form, despite the claim. It can be (η)-reduced to the term we are denoting as I. There is only one place where this affects the proofs above, as indicated in the paragraph after next.

But first let's double our money. The reader may have noticed that (η)-reduction never comes in anywhere in these proofs—it's always (β)-reduction. Furthermore, the numeral $\overline{1}$, in the intensional λ-calculus, IS in normal form. So, using the corresponding three syntactic theorems from just a few paragraphs above (whose proofs are actually somewhat easier in the intensional case), we have a perfectly good and standard proof that every recursive function is λ-definable in the intensional λ-calculus.

The only place where the non-normality of the numeral $\overline{1}$ is problematic, in the proofs above for the extensional λ-calculus, is in the proof of **VII-5.5**, where we go from asserting "$\approx$" to asserting "$\overline{\Sigma}$", when it happens that the right-hand side is $\overline{1}$. But just use the above result for the intensional λ-calculus to see that we can get defining terms for which these reductions exist. So that takes care of the small anomaly (and it did seem simplest to ignore it and to be a bit intellectually dishonest until now).

Sketch of the proof that λ-definable $\Longrightarrow \mathcal{RC}$.

This can be done following the pattern which we used much earlier in [**CM**] to show $\mathcal{BC} \Longrightarrow \mathcal{RC}$ (i.e. Babbage computable implies recursive). Let $A \in \Lambda$. Define functions $f_{n,A} : D_{n,A} \to \mathbf{N}$ as follows:

$$D_{n,A} \;:=\; \{\; \vec{v} \mid A\overline{v}_1 \cdots \overline{v}_n \approx \overline{\ell} \;\text{ for some } \ell \in \mathbf{N} \;\} \;\subset\; \mathbf{N}^n \;;$$

and since, by Church-Rosser, such an ℓ is unique (given $\vec{v}$), define

$$f_{n,A}(\vec{v}) \;:=\; \ell \;.$$

Every λ-definable function has the form $f_{n,A}$ for some (n, A) [though the function $f_{n,A}$ itself might not actually be definable using the term A, if A is not chosen so that $A\overline{v}_1 \cdots \overline{v}_n$ "loops" for all $\vec{v} \notin D_{n,A}$] . However, we prove that *all* $f_{n,A}$ are recursive.

In fact, as in the case of $\mathcal{BC}$-computable functions (but using lower-case names so as not to confuse things), one is able to write

$$f_{n,A}(\vec{v}) \;=\; \mathsf{print}(\min\{h \mid \mathsf{kln}(g_A, \vec{v}, h) = 1 \;\}) \;.$$

The functions kln and print will be defined below in such a way that it is *manifestly believable* that they are primitive recursive. This will use Gödel numbering for terms in Λ ; and g_A above is the Gödel number of A . To change "*believable*" to *true* is a tedious but unsubtle process, analogous to much earlier material, especially the proof of primitive recursiveness for KLN and $PRINT$. The proof here for kln and print will be omitted. Via the primitive recursiveness of kln and print, the recursiveness of $f_{n,A}$ follows immediately from the above display. [Note that this will also give a new proof of Kleene's normal form theorem for recursive functions, using the λ-calculus in place of **ATEN**.]

So here at least are the needed definitions. The argument using the definitions below works in the intensional λ-calculus, just as well as in our case, the extensional λ-calculus.

As for Gödel numbering of terms, define, somewhat arbitrarily, using the coding of finite strings of natural numbers from the Appendix before **IV-3** in **[CM]** :

$$G\ddot{o}d(x_i) =< 1, i > \;;\; G\ddot{o}d(AB) =< 2, G\ddot{o}dA, G\ddot{o}dB > \;;\; G\ddot{o}d(\lambda x_i \bullet A) =< 3, i, G\ddot{o}dA > \;.$$

Now define kln to be the relation for which

$$\mathsf{kln}(g, \vec{v}, h) = 1 \iff \exists A \in \Lambda \text{ satisfying the following :}$$

(i) $g \;=\; G\ddot{o}d(A)$;
(ii) $A\overline{v}_1 \cdots \overline{v}_n$ has a finite leftmost reduction leading to a term of the form $\overline{\ell}$;

(iii) h is the code of the history of that leftmost reduction :
that is, if the reduction is

$$A\overline{v}_1 \cdots \overline{v}_n = B_1 \ \overline{\succeq} \ B_2 \ \overline{\succeq} \ B_3 \ \overline{\succeq} \ \cdots \ \overline{\succeq} \ B_k = \overline{\ell} \ ,$$

then $h =< b_1, \cdots, b_k >$, where $b_i = G\ddot{o}d(B_i)$.

Finally $\mathsf{print} : \mathbf{N} \to \mathbf{N}$ may be suitably defined (see the next exercise) so that

$$\mathsf{print}(< b_1, \cdots, b_k >) = \ell \ \text{ if } \ b_k = G\ddot{o}d(\overline{\ell}) \ .$$

Exercises to define print.

Take $\overline{\ell}$ explicitly to be $\lambda x_2 \bullet (\lambda x_1 \bullet x_2{}^{\ell} x_1)$.

(i) Show that

$$G\ddot{o}d(\overline{3}) \ = \ < 3, 2, < 3, 1, < 2, < 1, 2 >, < 2, < 1, 2 >, < 2, < 1, 2 >, < 1, 1 >>>>>> \ .$$

(ii) Find a primitive recursive function Φ so that $\Phi(< 1, 1 >) = 0$ and $\Phi(< a, b, x >) = 1 + \Phi(x)$ for all a, b and x.

(iii) Define $\Psi(k) \ := \ \Phi(CLV(CLV(k, 3), 3))$. Prove : $\forall \ell, \ \Psi(G\ddot{o}d(\overline{\ell})) = \ell$.

(iv) Defining

$$\mathsf{print}(x) \ := \ \Psi(CLV(x, CLV(x, 0))) \ ,$$

check that you have a primitive recursive function with the property displayed just before the exercise.

Both the λ-calculus and the theory of combinators were originally developed as foundations for mathematics before digital computers were invented. They languished as obscure branches of mathematical logic until rediscovered by computer scientists. It is remarkable that a theory developed by logicians has inspired the design of both the hardware and software for a new generation of computers. There is an important lesson here for people who advocate reducing support for 'pure' research: the pure research of today defines the applied research of tomorrow.

•
•
•

$\cdots$ David Turner proposed that Schönfinkel and Curry's combinators could be used as machine code for computers for executing functional programming languages. Such computers could exploit mathematical properties of the λ-calculus $\cdots$

•
•
•

We thus see that an obscure branch of mathematical logic underlies important developments in programming language theory, such as:

(i) The study of fundamental questions of computation.

(ii) The design of programming languages.

(iii) The semantics of programming languages.

(iv) The architecture of computers.

Michael Gordon [**G**]

VII-6 Combinatorial Completeness and the Invasion of the Combinators.

Let Ω be a set with a binary operation, written as juxtaposition. We shall be discussing such objects a lot, and that's what we'll call them, rather than some subname of hemi-demi-semi-quasiloopoid/groupoid/algebraoid, or indeed applicative set, though the latter is suggestive of what we are trying to understand here. The set Ω^{Ω^n} consists of all functions of "n" variables from Ω, taking values in Ω, i.e. functions $\Omega^n \to \Omega$. It has a rather small subset consisting of those functions 'algebraically definable just using the operation'—for example,

$$(\omega_1, \omega_2, \omega_3) \mapsto (\omega_2((\omega_3\omega_3)(\nu\omega_1)))\omega_2 \quad ,$$

where ν is some fixed element of Ω. [We'll express this precisely below.] The binary operation Ω is said to be combinatorially complete if and only if every such "algebraically definable" function can be given as left multiplication by at least one element of Ω (i.e. 'is representable'). For example, there would have to be an element ζ such that, for all $(\omega_1, \omega_2, \omega_3)$ we have

$$\zeta\omega_1\omega_2\omega_3 \;=\; (\omega_2((\omega_3\omega_3)(\nu\omega_1)))\omega_2 \; .$$

We are using the usual convention here on the left, that is,

$$\zeta\omega_1\omega_2\omega_3 \;:=\; ((\zeta\omega_1)\omega_2)\omega_3 \; .$$

Note that the cardinality of Ω^{Ω^n} is strictly bigger than that of Ω except in the trivial case that Ω has only one element. So we could not have expected this definition to lead anywhere without some restriction (such as "algebraically definable") on which functions are supposed to be realizable by elements ζ .

We shall show quite easily in the next section that $\Lambda/\approx$ is combinatorially complete, where the binary operation is the one inherited from Λ. Then we introduce an ostensibly simpler set Γ, basically the combinators extended with variables. This produces the original combinatorially complete binary operation, namely $\Gamma/\sim$, due to Schonfinkel, who invented the idea. He didn't prove it this way, since the λ-calculus hadn't been invented yet, but we do it in the second section below by showing that the two binary operations $\Lambda/\approx$ and $\Gamma/\sim$ are isomorphic. So we don't really have two different examples here, but the new description using Γ is simpler in many respects.

Reduction algorithms for "$\sim$" in Γ have actually been used in the design of computers, when the desire is for a machine well adapted to functional programming languages [**G**].

Combinatorial Completeness of $\Lambda/\approx$.

First let's give a proper treatment of which functions we are talking about, that is, which are *algebraically definable*. The definition below is rather formal. It makes proving facts about these functions easy by induction on that inductive definition. However, for those not hidebound by some constructivist philosophy, there are simpler ways of giving the definition—"...the smallest set closed under pointwise multiplication of functions, and containing all the projections and all constant functions...". See the axiomatic construction of combinatorially complete binary operations near the beginning of Subsection VII-8 for more on this.

If Θ is any set, then the free binary operation generated by Θ, where we use $*$ to denote the operation, is the smallest set, $FREE(\Theta)$, of non-empty finite strings of symbols from $\Theta \cup \{ \ (\ , \ * \ , \ ) \ \}$ (a disjoint union of course!) such that each element of Θ is such a (length 1) string, and the set is closed under $(g, h) \mapsto (g{*}h)$. So it consists of strings such as $((\theta_3{*}((\theta_1{*}\theta_2){*}\theta_4)){*}\theta_3)$.

Now suppose given a binary operation on Ω as in the introduction, so here the operation is denoted by juxtaposing. Let $\{v_1, v_2, \cdots\}$ be a sequence of 'variables', all distinct, and disjoint from Ω, and, for each $n \geq 1$, take Θ to be the set $\Omega \ \cup \ \{v_1, v_2, \cdots, \ v_n\}$.

Now define a function

$$FUN = FUN_n \ : \ FREE(\Omega \ \cup \ \{v_1, v_2, \cdots, v_n\}) \ \rightarrow \Omega^{\Omega^n}$$

by mapping each element of Ω to the corresponding constant function; mapping each v_i to the ith projection function; and finally requiring that

$$FUN(A * B) \ = \ FUN(A)FUN(B) \ .$$

On the right-hand side, we're using the operation in Ω . (So actually FUN is the unique morphism of binary operations which acts as specified on the length 1 strings, the generators of the free binary operation, and where the binary operation in Ω^{Ω^n} is pointwise multiplication using the given operation in Ω.) Then the functions algebraically definable using that operation are

defined to be those which are in the image of the FUN_n for some $n \geq 1$. In the example beginning this section, the function given is

$$FUN_3 \; [\; (v_2 * ((v_3 * v_3) * (\nu * v_1))) * v_2 \;] \; .$$

So let's formalize the earlier definition:

Definition. The binary operation Ω is *combinatorially complete* if and only if, for all n and for all $f \in FREE(\Omega \cup \{ v_1 , v_2 , \cdots , v_n \})$, there is a $\zeta \in \Omega$ such that, for all $(\omega_1, \omega_2, \cdots, \omega_n)$ we have

$$\zeta\omega_1\omega_2 \cdots \omega_n \; = \; FUN_n(f)(\omega_1, \omega_2, \cdots, \omega_n) \; .$$

Theorem VII-6.1. *The binary operation on* $\Lambda/\approx$ *, induced by the juxtaposition operation on* Λ*, is combinatorially complete.*

Proof. Fix n, and proceed by induction on f in the definition above. The initial cases require us to find ζ's which work for the constant functions and for the projections, something which we did in the last section. For the inductive step, let $f = (g * h)$, where we assume that we have ζ_g and ζ_h which work for g and h respectively. But now the right-hand side in the definition above is just

$$FUN_N(g * h)(\omega_1, \omega_2, \cdots, \omega_n) \; =$$

$$FUN_N(g)(\omega_1, \omega_2, \cdots, \omega_n) FUN_N(h)(\omega_1, \omega_2, \cdots, \omega_n) \; =$$

$$(\zeta_g\omega_1\omega_2 \cdots \omega_n)(\zeta_h\omega_1\omega_2 \cdots \omega_n) \; .$$

So we must be able to solve the following equation for the unknown ζ, where ζ_g and ζ_h are fixed, and the equation should hold for all $(\omega_1, \omega_2, \cdots, \omega_n)$:

$$\zeta\omega_1\omega_2 \cdots \omega_n \; = \; (\zeta_g\omega_1\omega_2 \cdots \omega_n)(\zeta_h\omega_1\omega_2 \cdots \omega_n) \; .$$

Perhaps your initial reaction is to do this as just another application of the fixed-point operator; and that will work perfectly well, but is overkill, since the unknown occurs exactly zero times on the right-hand side. A straightforward solution is simply

$$\zeta \; = \; \lambda x_1 x_2 \cdots x_n \bullet (\zeta_g x_1 x_2 \cdots x_n)(\zeta_h x_1 x_2 \cdots x_n) \; .$$

This is a bit sloppy, confusing work in Λ with work in $\Lambda/\approx$. We should have said that ζ is the equivalence class of Z, where, if Z_g and Z_h are elements in the equivalence classes ζ_g and ζ_h , respectively, we define

$$Z \;=\; \lambda x_1 x_2 \cdots x_n \bullet (Z_g x_1 x_2 \cdots x_n)(Z_h x_1 x_2 \cdots x_n) \ .$$

Verification is a straightforward λ-calculation, which is a bit quicker if you apply **VII-1.2** .

Combinators.

Let S and K be two distinct symbols, disjoint from our original sequence $x_1, x_2, \cdots$ of distinct variables (which, though it won't come up directly, are quite distinct from the variables v_i of the last section).

Definition. The binary operation Γ is written as juxtaposition, and is defined to be the free binary operation generated by the set $\{S, K, x_1, x_2, \cdots\}$. Its elements are called by various names : *combinatory terms* [**Sö**][**Ko**], *combinations* [**Sö**], **CL-***terms* [**Ba**], *combinatorial expressions* [**Go**]. We won't need a name for them as individuals. The combinators are the elements of the subset of Γ which is the free binary operation generated by $\{S, K\}$. Thus combinators are just suitable strings of S's and K's and lots of brackets. We shall again use the convention that in Γ, the string $P_1 P_2 P_3 \cdots P_n$ really means $((\cdots((P_1 P_2)P_3)\cdot\cdots\cdot)P_n)$.

Finally, let $\sim$ be the equivalence relation on Γ generated by the following conditions, where we are requiring them to hold for *all* elements A, B, A_1, etc. in Γ :

$$SABC \sim (AC)(BC) \quad ; \quad KAB \sim A \quad ;$$
$$A_1 \sim B_1 \ \text{ and } \ A_2 \sim B_2 \quad \Longrightarrow \quad A_1 A_2 \sim B_1 B_2 \ ;$$

and

$$Ax \sim Bx \ \Longrightarrow \ A \sim B \ , \text{ if the variable } x \text{ appears in neither } A \text{ nor } B \ .$$

Remarks. (i) See the proof of **VII-6.12** below for a more explicit description of $\sim$.

(ii) Don't be fooled by the 'argument', that the last (extensionality) condition is redundant, which tells you to just multiply on the left by K and apply the second and third conditions : the point is that $K(Ax)$ is a lot different than KAx.

(iii) The requirement that x isn't in A or B cannot be removed from the last condition defining $\sim$. For we have $K(xx)x \sim xx$, but $K(xx) \not\sim x$. If the latter was false, it would follow from the results below in several ways that $K(BB) \sim B$ for any B. Taking $B = I = SKK$, it would follow that $KI \sim I$, and then multiplying on the right by A, that $I \sim A$ for any A. This would show that $\sim$ relates any two elements. That contradicts the main result below that the $\sim$-classes are in 1-1 correspondence with the $\approx$-classes from Λ : The Church-Rosser theorem tells us that there are tons of $\approx$-classes.

(iv) Apologies to the λ-experts for avoiding a number of arcane subtleties, both here and previously. In particular, whether to assume that last condition, or something weaker such as nothing at all, occupies a good deal of the literature. This is the great *intensionality versus extensionality* debate.

> To make the ascent from mathematics to logic is to pass from the object language to the metalanguage, or, as it might be said without jargon, to stop toying around and start believing in something ... A function could be a scheme for a type of process which would become definite when presented with an argument ... Two functions that are extensionally the same might be 'computed', however, by quite different processes ... The mixture of abstract objects needed would obviously have to be very rich, and I worry that it is a quicksand for foundational studies ... Maybe after sufficient trial and error we can come to agree that intensions have to be believed in, not just reconstructed, but I have not yet been able to reach that higher state of mind.
>
> Dana Scott ([**SHJM-ed**], pp. 157-162)

But, for the intended audience here, that debate is simply an unnecessary complication on first acquaintance with the subject, I believe.

Theorem VII-6.2. *The binary operation on* $\Gamma/\sim$ *, induced by the juxtaposition operation on* Γ*, is combinatorially complete.*

More examples arising from combinatorial completeness.

The top half of the following table is already familiar. On all lines, that the λ-version gives the effect is immediate. The reader might enjoy the following exercises :

(1) Show that the definition gives the effect; that is, work directly with combinator identities rather than the λ-calculus.

(2) Use the λ-versions of the ingredients in the definition column, and reduce that λ-term to normal form, which should be the entry in the 3rd column, up to re-naming bound variables.

It should be understood that the variables in each entry of the third column are distinct from each other.

combinator	definition	normal λ-version	effect
K		$\lambda xy \bullet x$	$K\alpha\beta \ \overline{\succeq} \ \alpha$
S		$\lambda xyz\bullet(xz)(yz)$	$S\alpha\beta\gamma \ \overline{\succeq} \ (\alpha\gamma)(\beta\gamma)$
I	SKK	$\lambda x \bullet x$	$I\alpha \ \overline{\succeq} \ \alpha$
B	$S(KS)K$	$\lambda xyz\bullet x(yz)$	$B\alpha\beta\gamma \ \overline{\succeq} \ \alpha(\beta\gamma)$
W	$SS(KI)$	$\lambda xy\bullet xyy$	$W\alpha\beta \ \overline{\succeq} \ \alpha\beta\beta$
C	$S(BBS)(KK)$	$\lambda xyz\bullet xzy$	$C\alpha\beta\gamma \ \overline{\succeq} \ \alpha\gamma\beta$
G	$B(BS)B$	$\lambda xyzw\bullet x(yw)(zw)$	$G\alpha\beta\gamma\delta \ \overline{\succeq} \ \alpha(\beta\delta)(\gamma\delta)$
E	$B(BW(BC))(BB(BB))$	$\lambda xyzw\bullet x(yz)(yw)$	$E\alpha\beta\gamma\delta \ \overline{\succeq} \ \alpha(\beta\gamma)(\beta\delta)$

Exercises. (i) Find a combinator whose normal λ-version is $\lambda xy \bullet yx$.
(ii) Try to find a combinator whose normal λ-version is the term

$$\underline{ADD} \ = \ \lambda uvxy \bullet (ux)(vxy) \ .$$

Recall that $\underline{ADD} \ \overline{k} \ \overline{\ell} \ \approx \ \overline{k+\ell}$, from the exercise before **VII-2.8**, where it had a different name. Referring to the second part of that exercise, find also a combinator for multiplication.

As mentioned earlier, the first proof of **VII-6.2** will not be direct. Rather we'll show that $\Gamma/\sim$ is isomorphic to $\Lambda/\approx$. That is, there is a bijective function between them which preserves the binary operations. Since combinatorial completeness is clearly invariant up to isomorphism, by **VII-6.1** this is all we need (and of course it gives us the extra information that we haven't, strictly speaking, produced a *new* example of combinatorial completeness).

To do this we need only define functions

$$\Psi : \Gamma \to \Lambda \quad \text{and} \quad \Phi : \Lambda \to \Gamma$$

such that the following five results hold :

VII-6.3. *If* $P \sim Q$ *then* $\Psi(P) \approx \Psi(Q)$.

VII-6.4. *If* $A \approx B$ *then* $\Phi(A) \sim \Phi(B)$.

VII-6.5. *For all* $P \in \Gamma$, *we have* $\Phi\Psi(P) \sim P$.

VII-6.6. *For all* $A \in \Lambda$, *we have* $\Psi\Phi(A) \approx A$.

VII-6.7. *For all* P *and* Q *in* Γ, *we have* $\Psi(PQ) \approx \Psi(P)\Psi(Q)$.

That this suffices is elementary general mathematics which the reader can work out if necessary. The first two give maps back and forth between the sets of equivalence classes, and the second two show that those maps are inverse to each other. The last one assures us that the maps are morphisms.

Definition of Ψ. Define it to be the unique morphism of binary operations which maps generators as follows : all the variables go to themselves; K goes to $\underline{K} := \underline{T} := \lambda xy \bullet x$; and S goes to $\underline{S} := \lambda xyz \bullet (xz)(yz)$.

Remarks. (i) Since Ψ, by definition, preserves the operations, there is nothing more needed to prove **VII-6.7** (which is understated—it holds with $=$, not just $\approx$.

(ii) The sub-binary operation of Λ generated by $\underline{S}$, $\underline{K}$ and all the variables is in fact freely generated by them. This is equivalent to the fact that Ψ is injective. But we won't dwell on this or prove it, since it seems not to be useful in establishing that the map induced by Ψ on equivalence classes is injective. But for concreteness, the reader may prefer to identify Γ with that subset of Λ, namely the image of Ψ . So you can think of the combinators as certain kinds of closed λ-expressions, closed in the sense of having no free variables.

The main job is to figure out how to simulate, in Γ, the abstraction operator in Λ.

Definition of $\mu x \bullet$ in Γ . For each variable x and each $P \in \Gamma$, define $\mu x \bullet P$ as follows, by induction on the structure of P:

$$\text{If } P \text{ is an atom other than } x, \text{ define } \mu x \bullet P \ := \ KP \ .$$

$$\text{Define } \mu x \bullet x \ := \ SKK \ := \ I \ .$$

$$\text{Define } \mu x \bullet (QR) \ := \mu x \bullet QR \ := \ S(\mu x \bullet Q)(\mu x \bullet R) \ .$$

Definition of Φ . This is again inductive, beginning with the atoms, i.e. variables x :

$$\Phi(x) := x \quad ; \quad \Phi(AB) := \Phi(A)\Phi(B) \quad ; \quad \Phi(\lambda x \bullet A) := \mu x \bullet \Phi(A) \ .$$

It should be (but seldom is) pointed out that the correctness of this definition depends on unique readability of the strings which make up the set Λ .

The first result is the mirror image of the last part of that definition.

VII-6.8. *For all $P \in \Gamma$, we have* $\Psi(\mu x \bullet P) \ \approx \ \lambda x \bullet \Psi(P)$.

Proof. Proceed by induction on P.

When P is the variable x, the left-hand side is

$$\Psi(\mu x \bullet x) \ = \ \Psi(SKK) \ = \ \underline{S}\ \underline{K}\ \underline{K}\ ;$$

whereas the right-hand side is $\lambda x \bullet x$. When applied to an arbitrary $B \in \Lambda$ these two 'agree' (as suffices even with B just a variable):

$$\underline{S}\ \underline{K}\ \underline{K}\ B \ \approx \ \underline{K}\ B\ (\underline{K}\ B) \ \approx \ B \approx \ (\lambda x \bullet x)B \ .$$

The middle $\approx$ is just **VII-2.1**(a), since $\underline{K} = \underline{T}$, and the first one is **VII-2.5**. [Notice the unimpressive fact that we could have defined I to be SKZ for *any* $Z \in \Gamma$.]

When P is an atom other than x, the left-hand side is

$$\Psi(KP) \ = \ \Psi(K)\Psi(P) \ = \ \underline{K}\ \Psi(P) \ .$$

But if x is not free in A [and it certainly isn't when we take $A = \Psi(P)$ here], for any B we have

$$\underline{K}\ A\ B \ \approx \ A \ \approx \ (\lambda x \bullet A)B \ .$$

Finally, when P is QR, where the result is assumed for Q and R, the left-hand side is

$$\Psi(S(\mu x \bullet Q)(\mu x \bullet R)) \ = \ \underline{S}\ \Psi(\mu x \bullet Q)\Psi(\mu x \bullet R) \ \approx \ \underline{S}\ (\lambda x \bullet \Psi(Q))(\lambda x \bullet \Psi(R)) \ .$$

The right-hand side is $\lambda x \bullet \Psi(Q)\Psi(R)$. Applying these to suitable B yields respectively (use **VII-2.5** again) :

$$\Psi(Q)^{[x \mapsto B]}\Psi(R)^{[x \mapsto B]} \quad \text{and} \quad (\Psi(Q)\Psi(R))^{[x \mapsto B]} \ ,$$

so they agree, completing the proof.

Any λ-expert reading the previous proof and next result will possibly find them irritating. We have made extensive use of extensionality in the last proof. But the results actually hold (and are very important in more encyclopædic versions of this subject) with '$\approx$' replaced by '$\approx_{\text{in}}$', where the latter is defined using only rules (α) and (β), and congruence [that is, drop rule (η)]. So an exercise for the reader is to find proofs of these slightly more delicate facts.

Proof of VII-6.6. Proceed by induction on the structure of A :
When A is a variable x, we have $\Psi\Phi(x) = \Psi(x) = x$.
When $A = BC$, where the result holds for B and C,

$$\Psi\Phi(BC) \;=\; \Psi(\Phi(B)\Phi(C)) \;=\; \Psi\Phi(B)\Psi\Phi(C) \;\approx\; BC \;.$$

Finally, when $A = \lambda x \bullet B$, where the result holds for B, using **VII-6.8**,

$$\Psi\Phi(\lambda x \bullet B) = \Psi(\mu x \bullet \Phi(B)) \;\approx\; \lambda x \bullet \Psi\Phi(B) \;\approx\; \lambda x \bullet B \;,$$

as required.

Proof of (most of) VII-6.5. Proceed by induction on the structure of P: The inductive step is trivial, as in the second case above.
When A is a variable, this is also trivial, as in the first case above.
When $A = K$, it is a straightforward calculation : For any B,

$$\Phi\Psi(K)B \;=\; \Phi(\underline{K})B \;=\; \Phi(\lambda x \bullet (\lambda y \bullet x))B \;=\; (\mu x \bullet (\mu y \bullet \Phi x))B \;=$$

$$(\mu x \bullet Kx)B \;=\; S(\mu x \bullet K)(\mu x \bullet x)B \;=\; S(KK)IB \;\sim$$

$$KKB(IB) \;\sim\; K(SKKB) \;\sim\; K(KB(KB)) \;\sim\; KB \;.$$

Taking B to be a variable, we get $\Phi\Psi(K) \;\sim\; K$, as required.
When $A = S$, there is undoubtedly a similar, but very messy, calculation. But the author doubts whether he will ever have the energy to write it out, or, having done so, much confidence that it is free of error. So we postpone the remainder of the proof until just after that of **VII-6.13** below. But we have the good luck that nothing between here and there depends on the

present result, **VII-6.5**. [The method there also gives an alternative to the above calculation for showing $\Phi\Psi(K) \sim K$.]

For λ-experts we are avoiding some delicate issues here by imposing extensionality. A quick calculation shows that $\Phi(\Psi(K)) = S(KK)I$ —in fact, just drop the B in the first two displayed lines above to see this—but $S(KK)I$ and K are not related under the equivalence relation '$\sim_{\text{in}}$' defined as with '$\sim$' except that extensionality (the last condition) is dropped. One needs a version of the Church-Rosser theorem to prove this.

Proof of VII-6.3. By the definition of $\sim$, and basics on equivalence relations (see the proof of **VII-6.12** ahead for what I mean by this, if necessary), it suffices to prove one fact for each of the four conditions generating the relation $\sim$, those facts being

$$\Psi(SABC) \approx \Psi(AC(BC)) \quad ; \quad \Psi(KAB) \approx \Psi(A) \quad ;$$

$$\Psi(A_1) \approx \Psi(B_1) \text{ and } \Psi(A_2) \approx \Psi(B_2) \implies \Psi(A_1A_2) \approx \Psi(B_1B_2) \ ;$$

and

$\Psi(Ax) \approx \Psi(Bx) \Rightarrow \Psi(A) \approx \Psi(B)$, if the variable x appears in neither A nor B.

The latter two are almost trivial, and the first two are simple consequences of earlier identities : Using **VII-2.1**(a),

$$\Psi(KAB) \ = \ \Psi(K)\Psi(A)\Psi(B) \ = \ \underline{T}\ \Psi(A)\Psi(B) \ \approx \ \Psi(A) \ ,$$

as required. Using **VII-2.5**,

$$\Psi(SABC) \ = \ \Psi(S)\Psi(A)\Psi(B)\Psi(C) \ = \ \underline{S}\ \Psi(A)\Psi(B)\Psi(C) \ \approx$$

$$\Psi(A)\Psi(C)(\Psi(B)\Psi(C)) \ = \ \Psi(AC(BC)) \ ,$$

as required.

The proof of **VII-6.4** seems to be somewhat more involved, apparently needing the following results.

VII-6.9. *If x isn't in P, then $\mu x \bullet P \ \sim \ KP$.*

Proof. Proceeding by induction on P, the atomic case doesn't include $P = x$, so we get equality, not just $\sim$. The inductive step goes as follows : For any T,

$$(\mu x \bullet QR)T \ = \ S(\mu x \bullet Q)(\mu x \bullet R)T \ \sim \ S(KQ)(KR)T \ \sim$$

$$KQT(KRT) \ \sim \ QR \ \sim \ K(QR)T \ ,$$

and so, $\mu x \bullet QR \ \sim \ K(QR)$, as required.

VII-6.10. *If $P \sim Q$ then $\mu x \bullet P \sim \mu x \bullet Q$.*

Proof. By the definition of $\sim$, and basics on equivalence relations, it suffices to prove one fact for each of the four conditions generating the relation $\sim$, those facts being

$$\mu x \bullet SABC \sim \mu x \bullet AC(BC) \quad ; \quad \mu x \bullet KAB \sim \mu x \bullet A \quad ;$$

$\mu x \bullet A_1 \sim \mu x \bullet B_1$ and $\mu x \bullet A_2 \sim \mu x \bullet B_2 \implies \mu x \bullet A_1 A_2 \sim \mu x \bullet B_1 B_2$;

and finally, if the variable z appears in neither A nor B,

$$\mu x \bullet Az \sim \mu x \bullet Bz \implies \mu x \bullet A \sim \mu x \bullet B \ .$$

However we first eliminate need to deal with the last of these when $z = x$ by proving the case of the entire result where x is not in P or Q. In that case, using **VII-6.9**,

$$\mu x \bullet P \ \sim \ KP \ \sim \ KQ \ \sim \ \mu x \bullet Q \ .$$

As for the last fact when $z \neq x$, for any R we have

$$(\mu x \bullet Az)R \ = \ S(\mu x \bullet A)(\mu x \bullet z)R \ \sim \ (\mu x \bullet A)R((\mu x \bullet z)R) \ \sim$$

$$(\mu x \bullet A)R(KzR) \ \sim \ (\mu x \bullet A)Rz \ .$$

The same goes with B replacing A, so taking R as a variable different from z and which is not in A or B, we cancel twice to get the result.

For the second fact above, we have, for any C,

$$(\mu x \bullet KAB)C \;=\; S(\mu x \bullet KA)(\mu x \bullet B)C \;\sim\; (\mu x \bullet KA)C((\mu x \bullet B)C) \;\sim$$

$$S(\mu x \bullet K)(\mu x \bullet A)C((\mu x \bullet B)C) \;\sim\; (\mu x \bullet K)C((\mu x \bullet A)C)((\mu x \bullet B)C) \;\sim$$

$$KKC((\mu x \bullet A)C)((\mu x \bullet B)C) \;\sim\; K((\mu x \bullet A)C)((\mu x \bullet B)C) \;\sim\; (\mu x \bullet A)C \;,$$

as suffices.

The first one is similar but messier.

The third fact is quick:

$$\mu x \bullet A_1A_2 \;=\; S(\mu x \bullet A_1)(\mu x \bullet A_2) \;\sim\; S(\mu x \bullet B_1)(\mu x \bullet B_2) \;=\; \mu x \bullet B_1B_2 \;.$$

VII-6.11. *The variables occurring in $\mu x \bullet P$ are exactly those other than x which occur in P.*

Proof. This is just a book-keeping exercise, by induction on P.

VII-6.12. *$R \sim P$ implies $R^{[x \to Q]} \sim P^{[x \to Q]}$*

Proof. A basis for this and a couple of earlier proofs is the following explicit description of the relation '$\sim$' :

$R \sim P \iff$ there is a finite sequence of ordered pairs of elements of Γ, whose last term is the pair (R, P), and each of whose terms has at least one of the following seven properties. It is :

(1) (A, A), for some A ; or

(2) (B, A), where (A, B) occurs earlier in the sequence; or

(3) (A, C), where both (A, B) and (B, C) occur earlier for some B; or

(4) $(KAB\ ,\ A)$, for some A and B ; or

(5) $(SABC\ ,\ (AC)(BC))$, for some A, B and C ; or

(6) $(A_1A_2\ ,\ B_1B_2)$, where both (A_1, B_1) and (A_2, B_2) occur earlier; or

(7) (A, B), where (Az, Bz) occurs earlier in the sequence, and where z is some variable not occurring in A or B.

[This is true because the condition above does define an equivalence relation which satisfies the conditions defining '$\sim$' , and any pair that is related under any relation which satisfies the conditions defining '$\sim$', is also necessarily related under this relation just above.]

Call any sequence of ordered pairs as above a *verification* for $R \sim P$.

Now proceed by contradiction, assuming that $R \sim P$ has a shortest possible verification among all pairs for which the result fails for some x and some Q—shortest in the sense of sequence length.

Then the pair (R, P) cannot have any of the forms as in (1) to (5) with respect to its "shortest verification", because $(R^{[x \to Q]}, P^{[x \to Q]})$ would have the same form (with respect to another verification in the case of (2) and (3), by "shortest").

It also cannot have the form in (6), since (again by "shortest") we could concatenate verifications for $A_1^{[x \to Q]} \sim B_1^{[x \to Q]}$ and $A_2^{[x \to Q]} \sim B_2^{[x \to Q]}$ to prove that $(A_1 A_2)^{[x \to Q]} \sim (B_1 B_2)^{[x \to Q]}$.

So (R, P) must have the form in (7), i.e. we have $(R, P) = (A, B)$ where (Az, Bz) occurs earlier in that shortest sequence, for some z not in A or B; and yet $A^{[x \to Q]} \not\sim B^{[x \to Q]}$.

Now necessarily $z \neq x$, since otherwise x does not occur in A or B, so

$$A^{[x \to Q]} \;=\; A \;\sim\; B = B^{[x \to Q]} \;.$$

Note that if w is a variable not occurring anywhere in a verification, applying $^{[y \to w]}$ for any y to all terms in the pairs in the verification will produce another verification.

Now choose the variable w different from x, so that w is not in Q nor in any term in pairs in the above "shortest verification". Applying $^{[z \to w]}$ to everything up to the (Az, Bz) term in that verification shows that $Aw \sim Bw$ by a derivation shorter than the "shortest" one above. Thus we have $(Aw)^{[x \to Q]} \sim (Bw)^{[x \to Q]}$; that is, $A^{[x \to Q]} w \sim B^{[x \to Q]} w$. But now we can just erase w since it doesn't occur in $A^{[x \to Q]}$ nor in $B^{[x \to Q]}$, yielding the contradiction $A^{[x \to Q]} \sim B^{[x \to Q]}$, i.e. $R^{[x \to Q]} \sim P^{[x \to Q]}$, and completing the proof.

[Of course this isn't really a proof by contradiction; rather it is one by induction on the length of the shortest verification for $P \sim Q$.]

VII-6.13. *For all $P \in \Gamma$, we have $(\mu x \bullet P)x \sim P$, and, more generally* (noting that 'okayness' of substitution is not an issue in Γ) *the following analogue of the (β)-rule:*

$$(\mu x \bullet P)Q \;\sim\; P^{[x \to Q]} \;.$$

[Note that the first part plus 'extensionality' quickly give the direct analogue in Γ for the (η)-rule from Λ, namely

$$\mu x \bullet Rx \ \sim \ R \qquad \text{if } x \text{ is not in } R \ .$$

To see this, just apply the left side to x.]

Proof. Proceeding by induction on P for the first identity, the atomic case when $P = x$ just uses $Ix \sim x$. When P is an atom other than x,

$$(\mu x \bullet P)x \ = \ KPx \ \sim \ P \ .$$

For the inductive step,

$$(\mu x \bullet QR)x \ = \ S(\mu x \bullet Q)(\mu x \bullet R)x \ \sim \ (\mu x \bullet Q)x((\mu x \bullet R)x) \ \sim \ QR \ .$$

Then the second identity follows using the first one, using **VII-6.12**, and using the previous fact in **VII-6.11** about x not occurring in $\mu x \bullet P$:

$$(\mu x \bullet P)Q \ = \ ((\mu x \bullet P)x)^{[x \to Q]} \ \sim \ P^{[x \to Q]} \ .$$

Completion of the proof of VII-6.5. To prove the remaining fact, namely $\Phi\Psi(S) \ \sim \ S$, first we show that $\Phi\Psi(S)BCD \ \sim \ SBCD$ for any B, C, D . This is straightforward, using **VII-6.13** three times :

$$\Phi\Psi(S)BCD \ = \ \Phi(\underline{S})BCD \ = \ (\mu x \bullet (\mu y \bullet (\mu z \bullet xz(yz))))BCD \ \sim$$

$$(\mu y \bullet (\mu z \bullet xz(yz)))^{[x \to B]}CD \ = \ (\mu y \bullet (\mu z \bullet Bz(yz)))CD \ \sim$$

$$(\mu z \bullet Bz(yz))^{[y \to C]}D \ = \ (\mu z \bullet Bz(Cz))D \ \sim$$

$$(Bz(Cz))^{[z \to D]} \ = \ BD(CD) \ \sim \ SBCD \ .$$

Now just take B, C, D as three distinct variables and cancel.

VII-6.14. *If y doesn't occur in Q and $x \neq y$, then*

$$(\mu y \bullet P)^{[x \to Q]} \ \sim \ \mu y \bullet (P^{[x \to Q]}) \ .$$

Proof. Proceeding by induction on P, the atomic cases are as follows:

$\underline{P = y}$: Both sides give I .

$\underline{P = x}$: Both sides give KQ, up to $\sim$. (Use **VII-6.9** .)

$\underline{P \text{ is any other atom}}$: Both sides give KP .

The inductive step uses nothing but definitions :

$$(\mu y \bullet TR)^{[x \to Q]} \;=\; (S(\mu y \bullet T)(\mu y \bullet R))^{[x \to Q]} \;=$$

$$S(\mu y \bullet T)^{[x \to Q]}(\mu y \bullet R)^{[x \to Q]} \;\sim\; S(\mu y \bullet T^{[x \to Q]})(\mu y \bullet R^{[x \to Q]}) \;=$$

$$\mu y \bullet (T^{[x \to Q]} R^{[x \to Q]}) \;=\; \mu y \bullet ((TR)^{[x \to Q]}) \; .$$

VII-6.15. *If x is not free in A, then it does not occur in $\Phi(A)$.*

Proof. This is an easy induction on A, using **VII-6.11** and definitions.

VII-6.16. *If $A^{[x \to B]}$ is okay, then $\Phi(A^{[x \to B]}) \;\sim\; \Phi(A)^{[x \to \Phi(B)]}$.*

Proof. Proceeding by induction on A, the atomic cases are as follows:

$\underline{A = x}$: Both sides give $\Phi(B)$.

$\underline{A = y \neq x}$: Both sides give $\Phi(y)$.

The inductive steps are as follows:

(I) $\underline{A = CD}$: Since $C^{[x \to B]}$ and $D^{[x \to B]}$ are both okay as long as $(CD)^{[x \to B]}$ is,

$$\Phi((CD)^{[x \to B]}) \;=\; \Phi(C^{[x \to B]})\Phi(D^{[x \to B]}) \;\sim\; \Phi(C)^{[x \to \Phi(B)]}\Phi(D)^{[x \to \Phi(B)]} \;=$$

$$(\Phi(C)\Phi(D))^{[x \to \Phi(B)]} \;=\; \Phi(CD)^{[x \to \Phi(B)]} \; .$$

(II) $\underline{A = \lambda x \bullet C}$: We have,

$$\Phi((\lambda x \bullet C)^{[x \to B]}) \;=\; \Phi(\lambda x \bullet C) \;=\; \mu x \bullet \Phi(C) \; ,$$

whereas, using **VII-6.11**,

$$(\Phi(\lambda x \bullet C))^{[x \to \Phi(B)]} \;=\; (\mu x \bullet \Phi(C))^{[x \to \Phi(B)]} \;=\; \mu x \bullet \Phi(C) \; .$$

(III) $\underline{A = \lambda y \bullet C \;\text{ for } y \neq x}$: The first '$\sim$' below uses the inductive hypothesis and **VII-6.10**. The second one uses **VII-6.14**, observing that, by **VII-6.15**, the variable y doesn't occur in $\Phi(B)$ because it is not free in B, the

latter being the case because $(\lambda y \bullet C)^{[x \to B]}$ is okay.

$$\Phi((\lambda y \bullet C)^{[x \to B]}) \;=\; \Phi(\lambda y \bullet (C^{[x \to B]})) \;=\; \mu y \bullet (\Phi(C^{[x \to B]})) \;\sim$$
$$\mu y \bullet (\Phi(C)^{[x \to \Phi(B)]}) \;\sim\; (\mu y \bullet \Phi(C))^{[x \to \Phi(B)]} \;=\; (\Phi(\lambda y \bullet C))^{[x \to \Phi(B)]} \,.$$

VII-6.17. *If y isn't in P, then $\mu x \bullet P \;\sim\; \mu y \bullet (P^{[x \to y]})$.*
[Note that this is the direct analogue in Γ for the (α)-rule from Λ.]

Proof. We leave this to the reader—a quite straightforward induction on P, using only definitions directly.

Proof of VII-6.4. Proceeding inductively, and looking at the inductive procedure in the definition of Λ first : If $A_1 \approx A_2$ and $B_1 \approx B_2$, then

$$\Phi(A_1 B_1) \;=\; \Phi(A_1)\Phi(B_1) \;\sim\; \Phi(A_2)\Phi(B_2) \;=\; \Phi(A_2 B_2) \,,$$

and

$$\Phi(\lambda x \bullet A_1) \;=\; \mu x \bullet \Phi(A_1) \;\sim\; \mu x \bullet \Phi(A_2) \;=\; \Phi(\lambda x \bullet A_2) \,,$$

using **VII-6.10** in this last line.

So the result will follow from three facts, corresponding to the three basic reduction laws in the definition of $\approx$:

$(\eta)'$ $\quad \Phi(\lambda x \bullet Ax) \;\sim\; \Phi(A)$ if x is not free in A .

$(\beta)'$ $\quad \Phi((\lambda x \bullet A)B) \;\sim\; \Phi(A^{[x \to B]})$ if $A^{[x \to B]}$ is okay .

$(\alpha)'$ $\quad \Phi(\lambda x \bullet A) \;\sim\; \Phi(\lambda y \bullet A^{[x \to y]})$ if y is not free in A and $A^{[x \to y]}$ is okay.

Twice below we use **VII-6.15**.

To prove $(\eta)'$, for any Q, we have

$$\Phi(\lambda x \bullet Ax)Q \;=\; (\mu x \bullet \Phi(A)x)Q \;\sim\; (\Phi(A)x)^{[x \to Q]} \;=\; \Phi(A)Q \,,$$

as suffices, using **VII-6.13** for the middle step.

To prove $(\beta)'$, the left-hand side is

$$(\mu x \bullet \Phi(A))\Phi(B) \;\sim\; \Phi(A)^{[x \to \Phi(B)]} \;\sim\; \Phi(A^{[x \to B]}) \,,$$

using **VII-6.13** and **VII-6.16** .

To prove $(\alpha)'$, the left-hand side is

$$\mu x \bullet \Phi(A) \ \sim \ \mu y \bullet (\Phi(A)^{[x \to y]}) \ \sim \ \mu y \bullet \Phi(A^{[x \to y]}) \ ,$$

which is the right-hand side, using, respectively, [**VII-6.17** plus y not occurring in $\Phi(A)$], and then using **VII-6.16** .

This completes the proof that the combinators do produce a combinatorially complete binary operation, not different from that produced by the λ-calculus (in our so-called extensional context).

Schonfinkel's original proof is different, and not hard to follow. Actually with all this intricacy laid out above, we can quickly give his direct proof that $\Gamma/\sim$ is combinatorially complete. More-or-less just copy the proof that $\Lambda/\approx$ is combinatorially complete, replacing '$\lambda x\bullet$' everywhere by '$\mu x\bullet$' , as follows.

For the constant function of 'n' variables with value P, pick distinct variables y_i not in P and let

$$\zeta \ = \ \mu y_1 \bullet \mu y_2 \bullet \cdots \bullet \mu y_n \bullet P \ .$$

For the ith projection, let $\zeta \ = \ \mu x_1 \bullet \mu x_2 \bullet \cdots \bullet \mu x_n \bullet x_i$.
For the inductive step, let

$$\zeta \ = \ \mu x_1 \bullet \mu x_2 \bullet \cdots \bullet \mu x_n \bullet (\zeta_g x_1 x_2 \cdots x_n)(\zeta_h x_1 x_2 \cdots x_n) \ ,$$

as in the proof of **VII-6.1** .

Schonfinkel's theorem is a bit more general than this, as below. The ideas in the proof of the substantial half are really the same as above, so the main thing will be to set up the machinery. We'll leave out details analogous to those in the build-up above.

Theorem VII-6.18. (Schonfinkel) *Let Ω be any binary operation* (written as juxtaposition with the usual bracket conventions). *Then Ω is combinatorially complete if and only if it has elements κ and σ such that, for all $\omega_i \in \Omega$, we have*

$$\kappa\omega_1\omega_2 \ = \ \omega_1 \qquad \textit{and} \qquad \sigma\omega_1\omega_2\omega_3 \ = \ (\omega_1\omega_3)(\omega_2\omega_3) \ .$$

For example, as we saw above, when $\Omega = \Gamma/\sim$, we could take κ and σ to be the equivalence classes $[K]_\sim$ and $[S]_\sim$, respectively.

Proof. Assume it is combinatorially complete, and with $F = FUN_2(v_1)$ and $G = FUN_3((v_1 * v_3) * (v_1 * v_3))$, let κ and σ respectively be corresponding elements ζ from the definition of combinatorial completeness. Thus, as required,

$$\kappa\omega_1\omega_2 \;=\; F(\omega_1,\omega_2) \;=\; \omega_1 \quad \text{and} \quad \sigma\omega_1\omega_2\omega_3 \;=\; G(\omega_1,\omega_2,\omega_3) \;=\; (\omega_1\omega_3)(\omega_2\omega_3)\;.$$

For the converse, here is some notation :

$$\Omega \;\subset\; \Omega_+^{(0)} = FREE(\Omega) \;\subset\; \Omega_+ = FREE(\Omega \cup \{v_1, v_2, \cdots\}) \;=\; \bigcup_{n\geq 0} \Omega_+^{(n)}\;,$$

where

$$\Omega_+^{(n)} \;:=\; FREE(\Omega \cup \{v_1, v_2, \cdots, v_n\})\;.$$

As before, the operation in Ω_+ will be written $*$, to contrast with juxtaposition in Ω (and the v_i's are distinct and disjoint from Ω) .

Note that morphisms $\chi : \Omega_+ \to \Omega$ are determined by their effects on the set of generators, $\Omega \cup \{v_1, v_2, \cdots\}$. We'll only consider those which map each element of Ω to itself, so when restricted to $\Omega_+^{(0)}$, all such χ agree. Furthermore, we have, with $A_i \in \Omega$,

$$FUN_n(f)(A_1, \cdots, A_n) \;=\; \chi(f^{[v_1\mapsto A_1][v_2\mapsto A_2]\cdots[v_n\mapsto A_n]})\;.$$

This is because, for fixed $(A_1, \cdots, A_n)$, each side is a morphism $\Omega_+^{(n)} \to \Omega$, "of f" , and they agree on generators.

Now, for variables $y = v_i$ and $P \in \Omega_+$, define $\mu y \bullet P \in \Omega_+$ inductively much as before :

$$\mu y \bullet P \;:=\; k * P \quad \text{for } P \in \Omega \text{ or if } P \text{ is a variable } v_j \neq y\;;$$

$$\mu y \bullet y \;:=\; s * k * k \;\;;$$

$$\mu y \bullet (Q * R) \;:=\; s * (\mu y \bullet Q) * (\mu y \bullet R) \;\;.$$

It follows as before that the variables occurring in $\mu y \bullet P$ are precisely those other than y in P. Also, define an equivalence relation on Ω_+ by

$$a \sim b \;\iff\; \forall \chi\;,\; \chi(a) = \chi(b)$$

(referring to χ which map elements of Ω by the identity map). Then we have, proceeding by induction on P as in **VII-6.13**,

$$(\mu y \bullet P) * y \ \sim \ P \ ,$$

and, substituting Q for y,

$$(\mu y \bullet P) * Q \ \sim \ P^{[y \to Q]} \ ,$$

and finally, by induction on n,

$$(\mu y_1 \bullet \mu y_2 \bullet \cdots \mu y_n \bullet P) * Q_1 * \cdots * Q_n \ \sim \ P^{[y_1 \to Q_1][y_2 \to Q_2] \cdots [y_n \to Q_n]} \ .$$

The proof is then completed easily : Given $f \in \Omega_+^{(n)}$, define

$$\zeta \ := \ \chi(\mu v_1 \bullet \mu v_2 \bullet \cdots \mu v_n \bullet f) \ \in \ \Omega \ .$$

Then, for all $A_i \in \Omega$, we have $A_i = \chi(A_i)$, so

$$\zeta A_1 \cdots A_n \ = \ \chi(\mu v_1 \bullet \mu v_2 \bullet \cdots \mu v_n \bullet f)\chi(A_1) \cdots \chi(A_n) \ =$$

$$\chi((\mu v_1 \bullet \mu v_2 \bullet \cdots \mu v_n \bullet f) * A_1 * \cdots * A_n) \ =$$

$$\chi(f^{[v_1 \to A_1][v_2 \to A_2] \cdots [v_n \to A_n]}) \ = \ FUN_n(f)(A_1, \cdots, A_n) \ ,$$

as required.

VII-7 Models for λ-Calculus, and Denotational Semantics.

This will be a fairly sketchy introduction to a large subject, mainly to provide some motivation and references. Then, in the last two subsections, we give many more details about a particular family of λ-models.

As far as I can determine, there is still not complete agreement on a single best definition of what is a 'model for the λ-calculus'. For example, not many pages along in **[Ko]**, you already have a choice of at least **ten** definitions, whose names correspond to picking an element from the following set and erasing the commas and brackets :

$$\{ \mathsf{proto} \ , \ \mathsf{combinatorial} \ , \ \mathsf{Meyer} \ , \ \mathsf{Scott} \ , \ \mathsf{extensional} \} \times \{ \mathsf{lambda} \} \times \{ \mathsf{algebra} \ , \ \mathsf{model} \} \ .$$

And that's only the beginning, as further along we have "categorical lambda models", etc., though here the first adjective refers to the method of construction, rather than an axiomatic definition. In view of all this intricacy (and relating well to the simplifications of the last subsection), we shall consider in detail only what are called extensional lambda models . The definition is quite simple : Such an object is any structure $(D\ ,\ \cdot\ ,\ \kappa\ ,\ \sigma)$, where " $\cdot$ " is a binary operation on the set D, which set has specified distinct elements κ and σ causing it to be combinatorially complete as in **VII-6.18**, and the operation must satisfy

(extensionality) *for all* α *and* β $\quad [(\forall\gamma\ \alpha\cdot\gamma = \beta\cdot\gamma) \implies \alpha = \beta\ ]$.

Notice that, using extensionality twice and then three times, the elements κ and σ are unique with respect to having their properties

$$\kappa\omega_1\omega_2 \ = \ \omega_1 \qquad \text{and} \qquad \sigma\omega_1\omega_2\omega_3 \ = \ (\omega_1\omega_3)(\omega_2\omega_3)\ .$$

So they needn't be part of the structure, just assumed to exist, rather like the identity element of a group. That is, just assume combinatorial completeness (along with extensionality), and forget about any other structure.

The other 10+ definitions referred to above are similar (once one becomes educated in the conventions of this type of model theory/equational-combinatorial logic, and can actually determine what is intended—but see the lengthy digression in small print beginning two paragraphs below). Effectively each replaces extensionality with a condition that is strictly weaker (and there are many interesting models that are not extensional, and which in some cases are apparently important for the application to denotational semantics.) So the reader should keep in mind that the extensional case, which we emphasize except in the small print just below, is not the entire story. Much more on the non-extensional case can be found in the references. Except for first reading the digression below, the clearest treatment I know is that in **[HS]**, if you want to learn more on models in general, for the λ-calculus and for combinators.

One missing aspect in our definition above is this : Where did the λ-abstraction operator get to, since, after all, it is the λ-calculus we are supposed to be modelling? Several of the other definitions address this directly. Below for several paragraphs we discuss a way of arriving at a general sort of definition of the term λ-*model*, a little different from others of which I

am aware, but easily seen to be mathematically equivalent to the definitions usually given. In particular, this will indicate how to get maps from the λ-calculus, Λ, into any model, and quite explicitly into extensional ones as above, one such map for each assignment of variables.

A (not entirely) Optional Digression on λ- models.

To remind you of a very basic point about sets and functions, there is a 1-1 *adjointness* correspondence as follows, where A^B denotes the set of all functions with (domain, codomain) the pair of sets (B, A) :

$$\begin{array}{ccc} A^{B\times C} & \longleftrightarrow & (A^C)^B \\ f & \vdash\!\!\mapsto & [b \mapsto [c \mapsto f(b,c)]] \\ [(b,c) \mapsto g(b)(c)] & \mapsto\!\!\dashv & g \end{array}$$

[Perhaps the λ-phile would prefer $\lambda bc \bullet f(b,c)$ and $\lambda(b,c) \bullet gbc$.] Taking $B = C = A$, this specializes to a bijection between the set of all binary operations on A and the set $(A^A)^A$. In particular, as is easily checked, the *extensional* binary operations on A correspond to the *injective* functions from A to A^A .

Let us go back to some of the vague remarks of Subsection VII-3, and try to puzzle out a rough idea of what the phrase "model for λ-calculus" ought to mean, as an object in ordinary naive set-theoretic mathematics, avoiding egregious violations of the axiom of foundation. We'll do this without requiring the reader to master textbooks on model theory in order to follow along. It seems to the author that the approach below, certainly not very original, is more natural than trying to force the round peg of λ-calculus into the square hole of something closely similar to models for $1^{\underline{\text{st}}}$order theories.

We want some kinds of sets, D, for which each element in D can play a second role as also somehow representing a function from D to itself. As noted, the functions so represented will necessarily form a very small subset, say $[D \!\rightharpoonup\! D]$, of D^D, very small relative to the cardinality of D^D.

The simplest way to begin to do this is to postulate a surjective function ϕ, as below :

$$\phi \;:\; D \;\to\; [D \!\rightharpoonup\! D] \;\subset\; D^D \;.$$

So we are given a function from D to the set of all its self-functions, we name that function's image $[D \!\rightharpoonup\! D]$, and use ϕ as the generic name for this surjective adjoint. It is the adjoint of a multiplication $D \times D \to D$, by the remarks in the first paragraph of this digression.

Now let $\Lambda^{(0)}$ be the set of all *closed terms* in Λ, those without free variables. A model, D, such as we seek, will surely involve at least a function $\Lambda^{(0)} \to D$ with good properties with respect to the structure of $\Lambda^{(0)}$ and to our intuitive notion of what that structure is supposed to be modelling; namely, function *application* and function *abstraction*. As to the former, the obvious thing is to have the map be a morphism between the binary operations on the two sets.

Because Λ is built by structural induction, it is hard to define anything related to Λ without using induction on structure. Furthermore, by first fixing, for each variable x, an element $\rho(x) \in D$, we'd expect there to be maps (one for each "assignment" ρ)

$$\rho_+ \ : \ \Lambda \to D \ ,$$

which all agree on $\Lambda^{(0)}$ with the one above. (So their restrictions to $\Lambda^{(0)}$ are all the same map.) These ρ_+'s should map each variable x to $\rho(x)$. Thinking about application and abstraction (and their formal versions in Λ), we are led to the following requirements.

$$\rho_+ \ : \ x \ \mapsto \ \rho(x) \ ;$$
$$\rho_+ \ : \ MN \ \mapsto \ \phi(\rho_+(M))(\rho_+(N)) \ ;$$
$$\rho_+ \ : \ \lambda x \bullet M \ \mapsto \ \psi(d \mapsto \rho_+^{[x \mapsto d]}(M)) \ .$$

The middle display is the obvious thing to do for a 'semantic' version of application. It's just another way of saying that ρ_+ is a morphism of binary operations.

But the bottom display needs plenty of discussion. Firstly, the assignment $\rho^{[x \mapsto d]}$ is the assignment of variables which agrees with ρ, except that it assigns $d \in D$ to the variable x. (This is a bit like substitution, so our notation reflects that, but is deliberately different, $^{[x \mapsto d]}$ rather than $^{[x:=d]}$). For the moment, let $\psi(f)$ vaguely mean "some element of D which maps under ϕ to f". Thus, the bottom display says that $\lambda x \bullet M$, which we intuit as "the function of x which M gives", should be mapped by ρ_+ to the element of D which is "somehow represented" (to quote our earlier phrase) by the function $D \to D$ which sends d to $\rho_+^{[x \mapsto d]}(M)$.

Inductively on the structure of M, it is clear from the three displays that, for all ρ, the values $\rho_+(M)$ are completely determined, at least once ψ is specified (if the displays really make sense).

And it is believable that $\rho_+(M)$ will be independent of ρ for closed terms M. In fact, one would expect to be able to prove the following :

$$\forall M, \forall \rho, \forall \rho' \ [\forall x \ [x \text{ is free in } M \ \Rightarrow \rho(x) = \rho'(x)] \implies \rho_+(M) = \rho'_+(M)] \ .$$

This clearly includes the statement about closed M.

We'd also want to prove that terms which are related under the basic relation $\approx$ (or rather its non-extensional version $\approx_{\text{in}}$) will map to the same element in D.

But before we get carried away trying to construct these proofs, there is one big problem with the bottom display in the triple specification of ρ_+'s values :

How do we know, on the right-hand side, that the function $d \mapsto \rho_+^{[x \mapsto d]}(M)$ is actually in the subset $[D \rightharpoonup D]$??

This must be dealt with by getting more specific about what the model (D, ϕ, ψ) can be, particularly about ψ . Playing around with the difficulty just above, we find that, besides wanting $(\phi \circ \psi)(f) = f$ for all $f \in [D \rightharpoonup D]$, we can solve the difficulty as long as (i) D is combinatorially complete, and (ii) the composite the other way, namely $\psi \circ \phi$, is in $[D \rightharpoonup D]$.

So now, after all this motivation, here is the definition of λ-model or model for the λ-calculus which seems the most natural to me :

Definition of λ-model. It is a structure $(D\ ;\ \cdot\ ;\ \psi)$ such that :
(i) $(D\ ;\ \cdot)$ is a combinatorially complete binary operation with more than one element ;
(ii) $\psi : [D \multimap D] \to D$ is a right inverse for the surjective adjoint $\phi : D \to [D \multimap D]$ of the multiplication in (i), such that $\psi \circ \phi \in [D \multimap D]$.

Before going further, we should explain how our earlier definition (on which these notes will entirely concentrate after this subsection) is a special case. As noted in the first paragraph of this digression, if the binary operation is extensional, then ϕ is injective. But then it is bijective, so it has a unique right inverse ψ, and this is a 2-sided inverse. But then the last condition is automatic, since the identity map of D is in $[D \multimap D]$ by combinatorial completeness. So an extensional, combinatorially complete, and non-trivial binary operation is automatically a λ-model in a unique way, as required.

It is further than we wish to go in the way of examples and proofs, but it happens to be an interesting fact that there exist combinatorially complete binary operations for which the number of ψ's as in (ii) of the definition is zero, and others where there are more than one, even up to isomorphism. In other words, λ-models are more than just combinatorially complete binary operations—there is extra structure whose existence and uniqueness is far from guaranteed.

Let's now state the theorem which asserts the existence of the maps ρ_+ talked about earlier.

Theorem. *Let $(D\ ;\ \cdot\ ;\ \psi)$ be a λ-model. Then there is a unique collection*

$$\{\ \rho_+ :\ \Lambda \to D \mid \rho : \{x_1, x_2, \cdots\} \to D\ \}$$

for which the following hold :
(1) $\rho_+(x_i) = \rho(x_i)$ *for all* i ;
(2) $\rho_+(MN) = \rho_+(M) \cdot \rho_+(N)$ *for all terms* M *and* N ;
(3) (i) *The map* $(d \mapsto \rho_+^{[x \mapsto d]}(M))$ *is in* $[D \multimap D]$ *for all* ρ, x, *and* M ; *and*
(ii) $\rho_+(\lambda x \bullet M) = \psi(d \mapsto \rho_+^{[x \mapsto d]}(M))$ *for all* x *and* M .

As mentioned earlier, the uniqueness of the ρ_+ is pretty clear from (1), (2) and (3)(ii). And the following can also be proved in a relatively straightforward manner, inductively on structure: the dependence of $\rho_+(M)$ on only the values which ρ takes on variables free in M, the invariance of ρ_+ with respect to the basic equivalence relation "$\approx$" on terms, and several other properties. But these proofs are not just one or two lines, particularly carrying out the induction to establish existence of the ρ_+ with property (3)(i). Combinatorial completeness has a major role. We won't give the proof, but the following example should help to make it clear how that induction goes.

Example. This illustrates how to reduce calculation of the ρ_+ to statement (1) of the theorem, using (2) and (3)(ii), and then why the three needed cases of (3)(i) hold.

$$\rho_+(\lambda xyz \bullet yzx) = \psi(d \mapsto \rho_+^{[x\mapsto d]}(\lambda yz \bullet yzx)) = \psi(d \mapsto \psi(c \mapsto \rho_+^{[x\mapsto d][y\mapsto c]}(\lambda z \bullet yzx)))$$

$$= \psi(d \mapsto \psi(c \mapsto \psi(b \mapsto \rho_+^{[x\mapsto d][y\mapsto c][z\mapsto b]}(yzx)))) = \psi(d \mapsto \psi(c \mapsto \psi(b \mapsto cbd))) \ .$$

But we want to see, as follows, why each application of ψ makes sense, in that it applies to a function which is actually in $[D \mathrel{-\!\!\!\rightsquigarrow} D]$. Using combinatorial completeness, choose elements $\epsilon, \zeta_1, \zeta_2$ and ζ_3 in D such that, for all p, q, r, s and t in D, we have

$$\epsilon p = (\psi \circ \phi)(p) \ \ ; \ \ \zeta_1 pqr = prq \ \ ; \ \ \zeta_2 pqrs = p(qsr) \ \ ; \ \ \zeta_3 pqrst = p(qrst) \ .$$

[Note that a neat choice for ϵ is $\psi(\psi \circ \phi)$.]

Now

$$b \mapsto cbd = \zeta_1 cdb = \phi(\zeta_1 cd)(b) \ ,$$

so the innermost one is okay; it is $\phi(\zeta_1 cd) \in [D \mathrel{-\!\!\!\rightsquigarrow} D]$. But then

$$c \ \ \mapsto \ \ \psi(b \mapsto cbd) = \psi(\phi(\zeta_1 cd)) = \epsilon(\zeta_1 cd) = \zeta_2 \epsilon \zeta_1 dc = \phi(\zeta_2 \epsilon \zeta_1 d)(c) \ ,$$

so the middle one is okay; it is $\phi(\zeta_2 \epsilon \zeta_1 d) \in [D \mathrel{-\!\!\!\rightsquigarrow} D]$. But then

$$d \ \ \mapsto \ \ \psi(c \mapsto \psi(b \mapsto cbd) = \psi(\phi(\zeta_2 \epsilon \zeta_1 d)) = \epsilon(\zeta_2 \epsilon \zeta_1 d) = \zeta_3 \epsilon \zeta_2 \epsilon \zeta_1 d = \phi(\zeta_3 \epsilon \zeta_2 \epsilon \zeta_1)(d) \ ,$$

so the outer one is okay; it is $\phi(\zeta_3 \epsilon \zeta_2 \epsilon \zeta_1) \in [D \mathrel{-\!\!\!\rightsquigarrow} D]$.

Meyer-Scott approach.

The element ϵ from the last example determines $\psi \circ \phi$, and therefore ψ, since ϕ is surjective. It is easy to show that, for all a and b in D ,

$$\epsilon ab = ab \qquad \text{and} \qquad [\forall c, \ ac = bc] \implies \epsilon a = \epsilon b \quad :$$

$$\epsilon \cdot a \cdot b = (\psi \circ \phi)(a) \cdot b = \phi((\psi \circ \phi)(a))(b) = (\phi \circ \psi \circ \phi)(a)(b) = (\phi)(a)(b) = a \cdot b \ .$$

$$[\forall c, \ a \cdot c = b \cdot c] \quad \text{i.e.} \ \ \phi(a) = \phi(b) \qquad \implies \qquad \psi(\phi(a)) = \psi(\phi(b)) \quad \text{i.e.} \ \ \epsilon \cdot a = \epsilon \cdot b \ .$$

Using these two facts, one can quickly recover the properties of ψ. Thus our definition of λ-model can be redone as a triple $(D \ ; \ \cdot \ ; \ \epsilon)$ with (ii) replaced by the two properties just above. A minor irritant is that ϵ is not unique. That can be fixed by adding the condition $\epsilon\epsilon = \epsilon$. So this new definition is arguably simpler than the one given, but I think less motivatable; but the difference between them is rather slight.

Scholium on the common approach.

The notation $\rho_+(M)$ is very non-standard. Almost every other source uses the notation

$$[\![M]\!]_\rho^{\mathcal{M}} \ \ \text{where} \ \ \mathcal{M} = (D \ ; \ \cdot) \ .$$

Both in logic and in denotational semantics (and perhaps elsewhere in computer science), the heavy square brackets $[\![\ ]\!]$ seem to be ubiquitous for indicating 'a semantic version of

the syntactic object inside the $[\![\ ]\!]$'s'. As mentioned earlier, the vast majority of sources for λ-*model* (and for weaker (more general) notions such as λ-*algebra* and *proto-*λ-*algebra*) express the concepts' definitions by lists of properties of the semantic function $[\![\ ]\!]$. And, of course, you'll experience a heavy dosage of "$\models$" and "$\vdash$". Again **[Hind-Seld]**, Ch. 11 (and Ch. 10 for models of combinatory logic), is the best place to start, esp. pp.112-122, for the list of properties, first the definition then derived laws.

Given such a definition, however weakly motivated, one can get to our definition—more quickly than the other way round (the theorem above embellished)—by defining ψ using

$$\psi(\phi(a)) \ := \ \rho_+^{[y\mapsto a]}(\lambda x \bullet yx) \ .$$

Note that $\lambda x \bullet yx \approx y$, but $\lambda x \bullet yx \not\approx_{\text{in}} y$, as expected from this. The properties of ρ_+ vaguely alluded to above assure one that

$$\phi(a) = \phi(b) \quad [\text{i.e. } \forall d,\ ad = bd] \quad \Longrightarrow \quad \rho_+^{[y\mapsto a]}(\lambda x \bullet yx) = \rho_+^{[y\mapsto b]}(\lambda x \bullet yx) \ ,$$

whereas $\rho_+^{[y\mapsto a]}(y) \neq \rho_+^{[y\mapsto b]}(y)$ clearly, if $a \neq b$!

The combinatorial completeness in our definition will then follow from Schonfinkel's theorem, noting that $\sigma := \rho_+(\lambda xyz\bullet(xz)(yz))$ and $\kappa := \rho_+(\underline{T})$ (for any ρ) have the needed properties.

Again, from the viewpoint which regards $[\![\ ]\!]$, or ρ_+, as primary, the element ϵ previous would be defined by $\epsilon := \rho_+(\lambda xy \bullet xy)$ for any ρ ;—hardly surprising in view of the law $\epsilon ab = ab$.

Finally, going the other way, a quick definition of ρ_+ for an *extensional* D is just the composite

$$\rho_+ \ = \ (\Lambda \ \xrightarrow{\Phi} \ \Gamma \ \xrightarrow{\rho_*} \ D) \ ,$$

where we recall from VII-6 that Γ is the free binary operation on $\{S, K, x_1, x_2, \cdots\}$ and the map Φ is the one inducing the isomorphism between $\Lambda/\approx$ and $\Gamma/\sim$, and here, ρ_* is defined to be the unique morphism of binary operations taking each x_i to $\rho(x_i)$, and taking S and K to the elements σ and κ from Schonfinkel's theorem (unique by extensionality).

Term model.

The set of equivalence classes $\Lambda/\approx_{\text{in}}$ is a combinatorially complete non-trivial binary operation. It is not extensional, but would be if we used "$\approx$" in place of "$\approx_{\text{in}}$". The canonical way to make it into a λ-model, using our definition here, is to define

$$\psi(\phi([M]_{\approx_{\text{in}}})) \ := \ [\lambda x \bullet Mx]_{\approx_{\text{in}}} \qquad \text{for any } x \text{ not occurring in } M \ .$$

It will be an exercise for you to check that this is well-defined, and satisfies (ii) in the definition of λ-*model*. Notice that, as it should, if we used "$\approx$" in place of "$\approx_{\text{in}}$", the displayed formula would just say that $\psi \circ \phi$ is the identity map.

What does a λ-model do?

(1) For some, using the Meyer-Scott or 'our' definition, it is sufficient that it provides an interesting genus of mathematical structure, one for which finding the 'correct' definition

had been a decades-long effort, and for which finding examples more interesting than the term models was both difficult and important. See the next subsection.

(2) For computer scientists, regarding the λ-calculus as a super-pure functional programming language, the functions $[\![\ ]\!]$ are the denotational semantics of that language. See the rest of this subsection, the table on p.155 of [**St**], and the second half of Subsection VII-9.

(3) For logicians, $[\![\ ]\!]$ provides the semantics for a putative form of proto-logic. Or, coming down from that cloud, one may think of $[\![\ ]\!]$ as analogous to the "Tarski definition of truth" in standard $1^{\underline{\text{st}}}$order logic. Again, read on !

Back to regular programming.

Why should one wish to produce models other than the term models? (There are at least four of those, corresponding to using " $\approx$ " or " $\approx_{\text{in}}$ ", and to restricting to closed terms or allowing terms with free variables. But the closed term models actually don't satisfy our definition above of λ-model, just one of the weaker definitions vaguely alluded to earlier.) As far as I can tell, besides the intrinsic mathematical interest, the requirements of denotational semantics include the need for individual objects inside the models with which one can work with mathematical facility. The syntactic nature of the elements in the term models dictate against this requirement. Further requirements are that one can manipulate with the models themselves, and construct new ones out of old with gay abandon. See the remarks below for more on this, as well as Subsection VII-9.

So we shall content ourselves, in the next subsection, 'merely' to go through Scott's original construction, which (fortunately for our pedagogical point of view) did produce *extensional* models. First I'll make some motivational and other remarks about the computer science applications. This remaining material also ties in well with some of the rather vague comments in VII-3.

The remainder of this subsection contains what I have gleaned from an unapologetically superficial reading of some literature on denotational semantics. To some extent, it may misreprepresent what has been really in the minds of the pioneers/practitioners of this art. I would be grateful for corrections/criticisms from any knowledgeable source. See Subsection VII-9 for some very specific such semantics of the language **ATEN** from the main part of this paper, and for the semantics of the λ-calculus itself.

We did give what we called the semantics of **BTEN** right after the syn-

tactic definition of that language. As far as I can make out, that was more like what the CSers would call an "operational semantics", rather than a denotational semantics. The reason is that we referred to a 'machine' of sorts, by talking about "bins" and "placing numbers in bins after erasing the numbers already there". That is a pretty weak notion of a machine. But to make the "meaning" of a language completely "machine independent" is one of the main criteria for an adequate denotational semantics.

See also the following large section on Floyd-Hoare logic. In it, we use a kind of operational semantics for several command languages. These assign, to a (command, input)-pair, the entire sequence of states which the pair generates. So that's an even more 'machine-oriented' form of operational semantics than the above input/output form. On the other hand, Floyd-Hoare logic itself is sometimes regarded as a form of semantic language specification, much more abstract than the above forms. There seems to have been (and still is?) quite a lot of controversy as to whether the denotational viewpoint is preferable to the above forms, especially in giving the definitions needed to make sense of questions of soundness and completeness of the proof systems in F-H logic. But we are getting ahead of ourselves.

Along with some of the literature on F-H logic, as well as some of Dana Scott's more explanatory productions, the two CS textbooks [**Go**] and [**Al**] have been my main sources. The latter book goes into the mathematical details (see the next subsection) somewhat more than the former.

In these books, the authors start with very small imperative programming languages : **TINY** in [**Go**], and "simple language" in Ch.5 of [**Al**]. These are clearly very close to **ATEN/BTEN** from early in this work. They contain some extras which look more like features in, say, **PASCAL**. Examples of "extras" are :

(1) the command "OUTPUT E" in [**Go**] ;

(2) the use of "BEGIN... END" in [**Al**], this being simply a more readable replacement for brackets ;

(3) the SKIP-command in [**Al**], which could be just $\mathsf{whdo}(0 = 1)(\text{any } C)$ or $x_0 \leftarrow\!\!\!\div\ x_0$ from **ATEN** .

So we may ask : which purely mathematical object corresponds to our mental image of a bunch of stored natural numbers, with the number called

v_i stored in bin number i ? That would be a function

$$\sigma \ : \ IDE \ \rightarrow \ VAL \ \ ,$$

where σ is called a state, IDE is the (syntactic) set of identifiers, and VAL is the (semantic) 'set' of values.

Now IDE for an actual command language might, for example, be the set of all finite strings of symbols from the 62-element symbol set consisting of the 10 decimal digits and the (26×2) upper- and lower-case letters of the Roman alphabet, with the proviso that the string begin with a letter (and with a few exclusions, to avoid words such as "while" and "do" being used as identifiers). For us in **BTEN**, the set IDE was simply the set $\{x_0, x_1, x_2, \cdots\}$ of variables. (In each case we of course have a countably infinite set. And in the latter case, there would be no problem in changing to a set of finite strings from a finite alphabet, for example the strings $x_{|||\cdots||}$. Actually, as long as the subscripts are interpreted as strings of [meaningless?] digits, as opposed to Platonic integers, we already have a set of finite strings from a finite alphabet.)

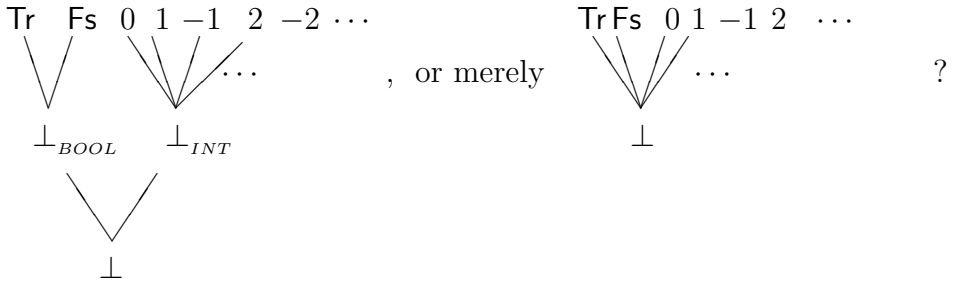
In referring above to VAL, we used " 'set' " rather than "set", because this is where λ-calculus models begin to come in, or what are called domains in this context. For us, VAL was just the set $\mathbf{N}= \{0, 1, 2, \cdots\}$ of all natural numbers. And σ was the function mapping $x_i \in IDE$ to $v_i \in VAL$. But for doing denotational semantics with a real, practical language, it seems to be essential that several changes including the following are made :
Firstly, the numbers might be more general, such as allowing all integers (negatives as well).
Secondly, VAL should contain something else, which is often called $\perp$ in this subject. It corresponds more-or-less to our use of "*err*" much earlier.
Thirdly, we should have a (rather weak) partial order $\sqsubseteq$ on the objects above (much more on this in the next subsection). As to the actual ordering here, other than requiring $b \sqsubseteq b$ for all b (part of the definition of *partial order*), we want $\perp \sqsubseteq c$ for all numbers $c \in VAL$ (but no other relation $\sqsubseteq$ between elements as above). So $\sqsubseteq$ is a kind of 'how much information?' partial order.
Finally, for a real command language, VAL would contain other semantic objects such as the truth values Tr and Fs. So, for example, in [**Al**], p.34, we see definitions looking like

$$Bv \ = \ INT + BOOL + Char + \cdots \ \ ,$$

and

$$VAL \;=\; Bv + [VAL + VAL] + [VAL \times VAL] + [VAL \rightharpoonup VAL] \quad .$$

Later there will be more on that last 'equation' and similar ones. Quite apart from the $\cdots$'s, **[Al]**'s index has no reference to $Char$ (though one might guess), and its index has double the page number, p.68, for Bv, which likely stands for 'basic values'. Also, even using only numbers and truth values, there are some decisions to make about the poset VAL ; for example, should it contain



In any case, it is crucial that we have at least the following :

(1) Some syntactically defined sets, often IDE, EXP and COM, where the latter two respectively are the sets of expressions and commands.

(2) Semantically defined 'domains' such as VAL and perhaps

$$STATE \;=\; [IDE \rightharpoonup VAL] \;,$$

where the right-hand side is a subposet of the poset of all functions from IDE to the underlying set of the structure VAL.

(3) Semantically defined functions

$$\mathcal{E} \;:\; EXP \;\rightarrow\; [STATE \rightharpoonup VAL] \;,$$

and

$$\mathcal{C} \;:\; COM \;\rightarrow\; [STATE \rightharpoonup STATE] \;.$$

The situation in **[Go]** is a bit messier (presumably of necessity) because $\perp$ is not used as a value, but separated out, called sometimes "unbound" and sometimes "error", but systematically.

Be that as it may, let us try to explain $\mathcal{E}$ and $\mathcal{C}$ in the case of our simple language **BTEN/ATEN**. Much more detail on this appears two subsections ahead.

Here EXP is the set of terms and quantifier-free formulae from the language of $1^{\underline{\text{st}}}$order number theory. Then $\mathcal{E}$ is that part of the Tarski definition of truth as applied to the particular interpretation, $\mathbf{N}$, of that $1^{\underline{\text{st}}}$order language which

(1) assigns to a term t, in the presence of a state $\underline{v}$, the natural number $t^{\underline{v}}$. That is, adding in the totally uninformative 'truth value' $\perp$,

$$\mathcal{E}[t](\underline{v}) \ := \ t^{\underline{v}} \ .$$

(But here we should add that $t^{\underline{v}} = \perp$ if any v_i, with x_i occurring in t, is $\perp$.)

(2) and assigns to a formula G, in the presence of a state $\underline{v}$, one the truth values. That is,

$$\mathcal{E}[G](\underline{v}) \ := \ \begin{cases} \mathsf{Tr} & \text{if } G \text{ is true at } \underline{v} \ ; \\ \mathsf{Fs} & \text{if } G \text{ is false at } \underline{v} \ ; \\ \perp & \text{if any } v_i, \text{ with } x_i \text{ free in } G, \text{ is } \perp \ . \end{cases}$$

Also, COM is the set of commands in **ATEN** or **BTEN**. And $\mathcal{C}$ assigns to a command C, in the presence of a state $\underline{v}$, the new state $||C||(\underline{v})$. See the beginning of Subsection VII-9 for more details.

So it appears that there is no great difference between the 'operational' semantics already given and this denotational semantics, other than allowing $\perp$ as a 'totally undefined number' or as a 'totally uninformative truth value'. But such a remark unfairly trivializes denotational semantics for several reasons. Before expanding on that, here is a question that experts presumably have answered elsewhere.

What are the main impracticalities of trying to bypass the whole enterprise of denotational semantics as follows ?

(1) Give, at the syntactic level, a translation algorithm of the relevant practical imperative programming language back into **ATEN**. For example, in the case of a self-referential command (i.e. recursive program), I have already done that in **IV-9** of this work. I can imagine that $GOTO$-commands would present some problems, but presumably not insuperable ones.

(2) Then use the simple semantics of **ATEN** to do whatever needs to be done, such as trying to prove that a program never loops, or that it is correct

according to its specifications, or that an F-H proof system is complete, or, indeed, such as implementing the language.

This is the sort of thing mentioned by Schwartz [**Ru-ed**] pp. 4-5, but reasons why it is not pursued seem not to be given there. Certainly one relevant remark is that, when one (as in the next section on F-H logic) generalizes **ATEN** by basing it on an arbitrary $1^{\underline{\text{st}}}$order language and its semantics on an arbitrary interpretation of that language, the translation as in (1) may no longer be possible. And so, even when the interpretation contains the natural numbers in a form which permits Gödel numbering for coding all the syntax, the translation might lack some kind of desirable naturality. And its efficiency would be a problem for sure.

This is becoming verbose, but there are still several matters needing explanation, with reference to why denotational semantics is of interest.

First of all, it seems clear that the richer 'extras' in real languages such as PASCAL and ALGOL60, as opposed to the really basic aspects already seen in these simpler languages, are where the indispensibility of denotational semantics really resides. (As we've emphasized, though these extras make programming a tolerable occupation, they don't add anything to what is programmable in principle.) Examples here would be those referred to in (1) just above—self-referential or recursive commands, and *GOTO*-statements (which are also self-referential in a different sense), along with declarations and calls to procedures with parameters, declarations of variables (related to our $\mathcal{B} - \mathcal{E}$-command in **BTEN**), declarations of functions, etc. Here we are completely ignoring parallelism and concurrent programming, which have become very big topics in recent years.

But I also get the impression that *mechanizing* the projects which use denotational semantics is a very central aspect here. See the last chapter of [**Al**], where some of it is made *executable*, in his phraseology. The mathematical way in which we had originally given the semantics of **BTEN** is inadequate for this. It's not *recursive* enough. It appears from [**Go**] and [**Al**], without being said all that explicitly, that one aspect of what is really being done in denotational semantics is to translate the language into a form of the λ-calculus, followed by perhaps some standard maps (like the ρ_+ earlier) of the latter into one or another "domain". So the metalanguage of the first half of this semantics becomes quite formalized, and (it appears to me), is a pure functional (programming?) language. (Perhaps other pure func-

tional programming languages don't need so much denotational semantics (beyond that given for the λ-calculus itself) since they are already in that form?)

For example, lines -5 and -6 of p. 53 of [**Al**] in our notation above and applied to **BTEN** become

$$\mathcal{C}[\mathbf{ite}(F)(C)(D)](\sigma) \;:=\; (\text{if } \mathcal{E}[F](\sigma) = \mathsf{Tr},\ \text{then } \mathcal{C}[C],\ \text{else } \mathcal{C}[D])(\sigma) \quad ,$$

for all formulas F, commands C and D, and states σ . At first, this appears to be almost saying nothing, with the if-then-else being semanticized in terms of if-then-else. But note how the right-hand side is a statement from the informal version of **McSELF**, in that what follows the "then" and "else" are not commands, but rather function values. In a more formal λ-calculus version of that right-hand side, we would just write a triple product, as was explained in detail at the beginning of Subsection VII-5.

Even more to the point here, lines -3 and -4 of p. 53 of [**Al**] in our notation above and applied to **BTEN** become

$$\mathcal{C}[\mathbf{whdo}(F)(C)](\sigma) \;:=\; (\text{if } \mathcal{E}[F](\sigma) = \mathsf{Tr}, \text{then } \mathcal{C}[\mathbf{whdo}(F)(C)] \circ \mathcal{C}[C], \text{else } Id)(\sigma) \quad ,$$

First notice how much more compact this is than the early definition in the semantics of **BTEN**. And again, the "if-then-else" on the right-hand side would be formalized as a triple product. But much more interestingly here, we have a self-reference, with the left-hand side appearing buried inside the right-hand side. So here we need to think about solving equations. The $\underline{Y}$-operator does that for us systematically in the λ-calculus, which is where the right-hand side resides, in one sense. The discussion of least fixed points ending the next subsection is clearly relevant. A theorem of Park shows that, at least in Scott's models from the next subsection, the same result is obtained from Tarski's 'least fixed points' operator in classical lattice theory (see **VII-8.12** ending the next subsection), as comes from the famous $\underline{Y}$-operator of Curry within λ-calculus.

We shall return to denotational semantics after the more purely mathematical subsection to follow on Scott's construction of λ-calculus models for doing this work.

VII-8 Scott's Original Models.

We wish to approach from an oblique angle some of the fundamental ideas of Dana Scott for constructing extensional models of the λ-calculus. There are many other ways to motivate this for mathematics students. In any case, presumably he won't object if this approach doesn't coincide exactly with his original thought processes.

One of several basic ideas here, in a rather vague form, is to employ some mathematical structure on A in order to construct such a model with underlying set A. We wish to define a (relatively small) subset of A^A, which we'll call $[A \multimap A]$, and a bijection between it and A itself. The corresponding extensional binary operation on A (via the adjointness discussed in the first paragraph of the digression in the last subsection) we hope will be combinatorially complete; that is, all its algebraically definable functions should be representable. And of course, representable implies algebraically definable almost by definition. To pick up on the idea again, can we somehow build a structure so that $[A \multimap A]$ turns out to be precisely the 'structure-preserving' functions from A to itself? If this were the case, and the structure-preserving functions satisfied a few simple and expected properties (with respect to composition particularly), then the proof of combinatorial completeness would become a fairly simple formality :

representable $\Longrightarrow$ algebraically definable

$\Longrightarrow$ structure-preserving $\Longrightarrow$ representable .

In the next several paragraphs, the details of how such an object would give an extensional model for the λ-calculus are given in an axiomatic style, leaving for later 'merely' the questions of what "structure" to use, and of self-reflection.

Details of essentially the category-theoretic approach.

Assume the following:

Data

Given a collection of 'sets with structure', and, for any two such objects, A and B, a subset $[A \multimap B] \subset B^A$ of 'structure preserving functions'. Given also a canonical structure on both the Cartesian product $A \times B$ and on $[A \multimap B]$.

Axioms

(1) The *ith* projection is in $[A_1 \times \cdots \times A_n \multimap A_i]$ for $1 \leq i \leq n$; in

particular, taking $n = 1$, the identity map, id_A, is in $[A \rightsquigarrow A]$.

(2) If $f \in [A \rightsquigarrow B]$ and $g \in [B \rightsquigarrow C]$ then the composition $g \circ f$ is always in $[A \rightsquigarrow C]$.

(3) The diagonal map δ is in $[A \rightsquigarrow A \times A]$, where $\delta(a) := (a, a)$.

(4) Evaluation restricts to a map ev in $[[A \rightsquigarrow B] \times A \rightsquigarrow B]$, where $ev(f, a) = f(a)$.

(5) If $f_1 \in [A_1 \rightsquigarrow B_1]$ and $f_2 \in [A_2 \rightsquigarrow B_2]$ then the $f_1 \times f_2$ is necessarily in $[A_1 \times A_2 \rightsquigarrow B_1 \times B_2]$, where $(f_1 \times f_2)(a_1, a_2) := (f_1(a_1), f_2(a_2))$.

(6) All constant maps are "structure preserving".

(7) The adjointness bijection, from $A^{B \times C}$ to $(A^C)^B$, maps $[B \times C \rightsquigarrow A]$ into $[B \rightsquigarrow [C \rightsquigarrow A]]$.

All this will be relatively easy to verify, once we've chosen the appropriate "structure" and "structure preserving maps", to make the following work, which is more subtle. That choosing will also be motivated by the application to denotational semantics.

Self-reflective Object

Now suppose given one of these objects A, and a mutually inverse pair of bijections which are structure preserving :

$$\phi \in [A \rightsquigarrow [A \rightsquigarrow A]] \quad \text{and} \quad \psi = \phi^{-1} \in [[A \rightsquigarrow A] \rightsquigarrow A] \ .$$

We want to show how such a "self-reflective" object may be made canonically into an extensional combinatorially complete binary operation.

Define the multiplication in A from ϕ, using adjointness—see the first paragraph of the digression in the last subsection—and so it is the following composite :

$$\begin{array}{rccccc} mult \ : & A \times A & \stackrel{\phi \times id}{\longrightarrow} & [A \rightsquigarrow A] \times A & \stackrel{ev}{\longrightarrow} & A \\ & (x, y) & \mapsto & (\phi(x), y) & \mapsto & \phi(x)(y) := x \cdot y \end{array}$$

By (1),(2),(4) and (5), this map, which we'll name $mult$, is in $[A \times A \rightsquigarrow A]$.

Now $[A \rightsquigarrow A]$ contains id_A and all constant maps, and is closed under pointwise multiplication of functions as follows :

$$\begin{array}{ccccccc} A & \xrightarrow{\delta} & A \times A & \xrightarrow{f \times g} & A \times A & \xrightarrow{mult} & A \\ x & \mapsto & (x,x) & \mapsto & (f(x), g(x)) & \mapsto & f(x) \cdot g(x) \end{array}$$

The fact that ϕ is injective implies that *mult* is extensional, as we noted several times earlier. First we shall check the 1-variable case of combinatorial completeness of *mult*.

Definitions. (More-or-less repeated from much earlier.)

Say that $f \in A^A$ is *1-representable* when there is a $\zeta \in A$ such that $f(a) = \zeta \cdot a$ for all $a \in A$.

Define the set of *1-algebraically definable* functions in A^A to be the smallest subset of A^A containing id_A and all constant functions, and closed under pointwise multiplication of functions.

1-combinatorial completeness of A is the fact that the two notions just above coincide.

Its proof is now painless in the form

1-representable $\Longrightarrow$ 1-algebraically definable

$\Longrightarrow$ structure-preserving $\Longrightarrow$ 1-representable .

The first implication is because a 1-representable f as in the definition is the pointwise multiplication of (the constant function with value ζ) times (the identity function).

The second implication is the fact noted above that the set $[A \rightharpoonup\!\!\!\!\!\sim\ A]$ is an example of a set containing id_A and all constant functions, and closed under pointwise multiplication of functions.

(Of course, the equation in the definition of 1-representability can be rewritten as $f = \phi(\zeta)$, so the equivalence of 1-representability with structure preserving becomes pretty obvious.)

The third implication goes as follows :

Given $f \in [A \rightharpoonup\!\!\!\!\!\sim\ A]$, define ζ to be $\psi(f)$. then

$$\zeta \cdot a = \psi(f) \cdot a = \phi(\psi(f))(a) = f(a) \ ,$$

as required.

Now we shall check the 2-variable case of combinatorial completeness, and leave the reader to pump this up into a proof for any number of variables.

Definitions.

Say that $f \in A^{A\times A}$ is *2-representable* when there is a $\zeta \in A$ such that $f(b,c) = (\zeta \cdot b) \cdot c$ for all b and c in A .

Define the set of *2-algebraically definable* functions in $A^{A\times A}$ to be the smallest subset of $A^{A\times A}$ containing both projections and all constant functions, and closed under pointwise multiplication of functions.

2-combinatorial completeness of A is the fact that the two notions just above coincide.

Its proof is much as in the 1-variable case :

2-representable $\Longrightarrow$ 2-algebraically definable

$\Longrightarrow$ structure-preserving $\Longrightarrow$ 2-representable .

The first implication is because a 2-representable f as in the definition is a suitably sequenced pointwise multiplication of the constant function with value ζ and the two projections.

The second implication is the fact that the set $[A \times A \multimap A]$ is an example of a set containing the projections and all constant functions, and closed under pointwise multiplication of functions. The latter is proved by composing:

$$A \times A \quad \stackrel{\delta}{\longrightarrow} \quad (A \times A) \times (A \times A) \quad \stackrel{f\times g}{\longrightarrow} \quad A \times A \quad \stackrel{mult}{\longrightarrow} \quad A$$

$$(b,c) \quad \mapsto \quad ((b,c),(b,c)) \quad \mapsto \quad (f(b,c),g(b,c)) \quad \mapsto \quad f(b,c)\cdot g(b,c)$$

The third implication goes as follows :

If $f \in [A \times A \multimap A]$, the composite $\psi \circ adj(f)$ is in $[A \multimap A]$, using axiom (7) for the first time. By the part of the 1-variable case saying that *structure preserving* implies *1-representable*, choose ζ so that, for all $b \in A$, we have $\psi(adj(f)(b)) = \zeta \cdot b$. Then

$$\zeta \cdot b \cdot c = \psi(adj(f)(b)) \cdot c = \phi(\psi(adj(f)(b)))(c) = adj(f)(b)(c) = f(b,c) \ ,$$

as required.

So now we must figure out what kind of structure will work, and how we might produce a self-reflective object.

The next idea has already occurred in the verbose discussion of denotational semantics of the previous subsection : the structure referred to above

maybe should be a partial order with some extra properties (so that, in the application, it intuitively coincides with 'comparing information content').

Now I believe (though it's not always emphasized) that an important part of Scott's accomplishment is not just to be first to construct a λ-model (and do so by finding a category and a self-reflective object in it), but also to show how to start with an individual from a rather general species of posets

(in the application, from { numbers , truth values , $\perp$ } at least) ,
and show how to embed it (as a poset) into an extensional λ-model.

So let's start with any poset $(D, \sqsubseteq)$ and see how far we can get before having to impose extra properties. Recall that the definition of *poset* requires

(i) $a \sqsubseteq b$ and $b \sqsubseteq c$ implies $a \sqsubseteq c$; and

(ii) $a \sqsubseteq b$ and $b \sqsubseteq a$ if and only if $a = b$.

Temporarily define $[D \multimap D]$ to consist of all those functions f which preserve order; that is

$$d \sqsubseteq e \implies f(d) \sqsubseteq f(e) \ .$$

More generally this defines $[D \multimap E]$ where E might not be the same poset as D.

Now $[D \multimap D]$ contains all constant functions, so we can embed D into $[D \multimap D]$ by $\phi_D : d \mapsto (d' \mapsto d)$. This $\phi_D : D \to [D \multimap D]$ will seldom be surjective. It maps each d to the constant function with value d.

Next suppose that D has a minimum element called $\perp$. Then we can map $[D \multimap D]$ back into D by $\psi_D : f \mapsto f(\perp)$. It maps each function to its minimum value.

It is a trivial calculation to show that $\psi_D \circ \phi_D = id_D$, the identity map of D. By definition, this shows D to be a *retract* of $[D \multimap D]$. (Note how this is the reverse of the situation in the digression of the last subsection, on the general definition of λ-model, where $[D \multimap D]$ was a retract of D.) In particular, ϕ_D is injective (obvious anyway) and ψ_D is surjective.

The set $[D \multimap D]$ itself is partially ordered by

$$f \sqsubseteq g \quad \iff \quad \forall d \ , \ f(d) \sqsubseteq g(d) \ .$$

This actually works for the set of *all* functions from any set to any poset.

How do ϕ_D and ψ_D behave with respect to the partial orders on their domains and codomains? Very easy calculations show that they do preserve the orders, and thus we have

$$\phi_D \in [D \multimap [D \multimap D]] \quad \text{and} \quad \psi_D \in [[D \multimap D] \multimap D] \ .$$

So D is a retract of $[D \multimap D]$ *as a poset*, not just as a set. But $[D \multimap D]$ is usually too big—the maps above are not inverses of each other, only injective and surjective, and one-sided inverses of each other, as we already said.

Now here's another of Scott's ideas : whenever you have a retraction pair $D \overset{\hookrightarrow}{\twoheadleftarrow} E$, you can automatically produce another retraction pair

$$[D \multimap D] \quad \overset{\hookrightarrow}{\twoheadleftarrow} \quad [E \multimap E] \quad .$$

If (ϕ, ψ) is the first pair and (ϕ', ψ') the second pair, then the formulae defining the latter are

$$\phi'(f) \;:=\; \phi \circ f \circ \psi \quad \text{and} \quad \psi'(g) \;:=\; \psi \circ g \circ \phi \quad .$$

Again a very elementary calculation shows that this is a retraction pair.

[As an aside which is relevant to your further reading on this subject, notice how what we've just done is purely 'arrow-theoretic' : the only things used are associativity of composition and behaviour of the identity morphisms. It's part of *category theory*. In fact Lambek has, in a sense, identified the theory of combinators, and of the typed and untyped λ-calculi, with the theory of *cartesian closed categories*. The "closed" part is basically the situation earlier with the axioms (1) to (7) where the set of morphisms between two objects in the category is somehow itself made into an object in the category, an object with good properties.]

Now it is again entirely elementary to check that, for the category of posets and order-preserving maps, the maps ϕ' and ψ' do in fact preserve the order.

Back to the earlier situation in which we have the poset D as a retract of the poset $E = [D \multimap D]$, the construction immediately above can be iterated : Define

$$D_0 := D \quad , \quad D_1 := [D_0 \multimap D_0] \quad , \quad D_2 := [D_1 \multimap D_1] \quad , \quad \text{etc. } \cdots \; .$$

Then the initial retraction pair $D_0 \overset{\hookrightarrow}{\twoheadleftarrow} D_1$ gives rise by the purely category-theoretic construction to a sequence of retraction pairs $D_n \overset{\hookrightarrow}{\twoheadleftarrow} D_{n+1}$. We'll denote these maps as (ϕ_n, ψ_n) .

Ignoring the surjections for the moment, we have

$$D_0 \overset{\phi_0}{\hookrightarrow} D_1 \overset{\phi_1}{\hookrightarrow} D_2 \overset{\phi_2}{\hookrightarrow} \cdots \quad .$$

Wouldn't it be nice to be able to 'pass to the limit', producing a poset D_∞ as essentially a union. And then to be able to just set $n = \infty = n+1$, and say that we'd get canonical *bijections* back-and-forth between D_∞ and $[D_\infty \multimap D_\infty]$??? (The first ∞ is $n+1$, and the second one is n, so to speak.) After all, that was the original objective here !! In fact, going back to some of the verbosities in Subsection VII-3, this looks as close as we're likely to get to a mathematical situation in which we have a non-trivial structure D_∞ which can be identified in a natural way with its set of (structure-preserving!) self-maps. The binary operation on D_∞ would of course come from the adjointness discussed at the beginning of this subsection; that is,

$$a_\infty \cdot b_\infty \ := \ (\phi_\infty(a_\infty))(b_\infty) \quad ,$$

where ϕ_∞ is the isomorphism from D_∞ to $[D_\infty \multimap D_\infty]$.

Unfortunately, life isn't quite that simple; but really not too complicated either. The Scott construction which works is not to identify D_∞ as some kind of direct limit of the earlier displayed sequence—thinking of the ϕ's as actual inclusions, that would be a nested union of the D_n's—but rather to define D_∞ as the inverse limit, using the sequence of surjections

$$D_0 \ \stackrel{\psi_0}{\twoheadleftarrow} \ D_1 \ \stackrel{\psi_1}{\twoheadleftarrow} \ D_2 \ \stackrel{\psi_2}{\twoheadleftarrow} \ \cdots \quad .$$

To be specific, let

$$D_\infty \ := \ \{ \ (d_0, d_1, d_2, \cdots) \mid \text{for all } i, \text{ we have } d_i \in D_i \text{ and } \psi_i(d_{i+1}) = d_i \ \} \quad .$$

Partially order D_∞ by

$$(d_0, d_1, d_2, \cdots) \sqsubseteq (e_0, e_1, e_2, \cdots) \iff \text{for all } i, \text{ we have } d_i \sqsubseteq e_i \quad .$$

This is easily seen to be a partial order (even on the set of *all* sequences, i.e. on the infinite product $\Pi_{i=0}^{\infty} D_i$).

[Notice that, as long as D_0 has more than one element, the set D_∞ will be uncountably infinite in cardinality. So despite this all being motivated by very finitistic considerations, it has drawn us into ontologically sophisticated mathematics.]

And there's another complication, but one of a rather edifying nature (in my opinion). In order to make the following results actually correct, we need

to go back and change *poset* to complete lattice and *order preserving map* to continuous map. The definitions impose an extra condition on both the objects and the morphisms.

Definition. A *complete lattice* is a poset $(D, \sqsubseteq)$, where every subset of D has a *least upper bound*. Specifically, if $A \subset D$, there is an element $\ell \in D$ such that

(i) $a \sqsubseteq \ell$ for all $a \in A$; and

(ii) for all $d \in D$, if $a \sqsubseteq d$ for all $a \in A$, then $\ell \sqsubseteq d$.

It is immediate that another least upper bound m for A must satisfy both $\ell \sqsubseteq m$ and $m \sqsubseteq \ell$, so the least upper bound is unique. We shall use $\sqcup A$ to denote it. In particular, a complete lattice always has a least element $\sqcup\emptyset$, usually denoted $\perp$.

Definition. A function $f : D \to E$ between two complete lattices is called *continuous* if and only if $f(\sqcup A) = \sqcup f(A)$ for all *directed subsets* A of D, where $f(A)$ is the usual image of the subset under f. Being *directed* means that, for any two elements a and b of A, there is a $c \in A$ with $a \sqsubseteq c$ and $b \sqsubseteq c$.

It follows for continuous f that $f(\perp_D) = \perp_E$. Also, such an f preserves order, using the fact that, if $x \sqsubseteq y$, the $x \sqcup y := \sqcup\{x, y\} = y$. If f is bijective, we call it an *isomorphism*. Its inverse is automatically continuous.

Definition. The set $[D \multimap E]$ is defined to consist of all continuous maps from D to E.

Scott's Theorem. *Let D be any complete lattice. Then there is a complete lattice D_∞ which is isomorphic to $[D_\infty \multimap D_\infty]$, and into which D can be embedded as a sublattice.*

For the theorem to be meaningful, the set $[D_\infty \multimap D_\infty]$ is made into a complete lattice as indicated in the first exercise below. In view of that exercise, this theorem is exactly what we want, according to our elementary axiomatic development in the last several pages.

For Scott's theorem, we'll proceed to outline two proofs in the form of sequences of exercises. First will be a rather "hi-fallutin' " proof, not much different than Lawvere's suggestion in **[La-ed]** p.179, but less demanding of sophistication about categories on the reader's part (as opposed to *categorical sophistication*, which must mean sophistication about one and only one thing,

up to isomorphism!).

Both proofs use the following exercise.

Ex. VII-8.1. (a) Verify the seven axioms for the category of complete lattices and continuous maps, first verifying (and doing part (b) below simultaneously) that $D \times E$ and $[D \mathrel{-\!\!\!\rightsquigarrow} E]$ are complete lattices whenever D and E are, using their canonical orderings as follows :

$$(d,e) \sqsubseteq_{D\times E} (d',e') \iff d \sqsubseteq_D d' \text{ and } e \sqsubseteq_E e' ;$$
$$f \sqsubseteq_{[D\rightsquigarrow E]} g \iff f(d) \sqsubseteq_E g(d) \text{ for all } d \in D .$$

The subscripts on the " $\sqsubseteq$ *" will be omitted in future, as they are clear from the context, and the same for the subscripts on the "* $\sqcup$ *" below.*

(b) Show also that the least upper bounds in these complete lattices are given explicitly by

$$\sqcup_{D\times E} A = (\sqcup_D p_1(A), \sqcup_E p_2(A)) \quad \text{where the } p_i \text{ are the projections, and}$$
$$(\sqcup_{[D\rightsquigarrow E]} A)(x) = \sqcup_E \{f(x) | f \in A\} .$$

(c) Verify also that all the maps ϕ_n and ψ_n are continuous.

(d) Show that D_∞ and $\prod_{i=0}^{\infty} D_i$ are complete lattices, and the inclusion of the former into the latter is continuous.

(e) Show that all the maps $\theta_{ab} : D_a \to D_b$ are continuous, including the cases when some subscripts are ∞, where the definitions are as follows (using θ so as not to show any favouritism between ϕ and ψ) :

When $a = b$, use the identity map.

When $a < b < \infty$, use the obvious composite of ϕ_i's.

When $b < a < \infty$, use the obvious composite of ψ_i's.

Let $\theta_{\infty,b}(d_0, d_1, d_2, \cdots) := d_b$.

Let $\theta_{a,\infty}(x) := (\theta_{a,0}(x), \theta_{a,1}(x), \cdots, \theta_{a,a-1}(x), x, \theta_{a,a+1}(x), \cdots)$.

Note that this last element *is* in D_∞ . See also **VII-8.6** .

Ex. VII-8.2. Show that D_∞ satisfies the 'arrow-theoretic' definition of being 'the' inverse limit (in the category of complete lattices and continuous maps) of the system

$$D_0 \xleftarrow{\psi_0} D_1 \xleftarrow{\psi_1} D_2 \xleftarrow{\psi_2} \cdots$$

with respect to the maps $\theta_{\infty n} : D_\infty \to D_n$. That is, given a complete lattice E and continuous maps $\eta_n : E \to D_n$ such that $\eta_n = \psi_n \circ \eta_{n+1}$ for all n, there is a unique continuous map $\eta_\infty : E \to D_\infty$ such that $\eta_n = \theta_{\infty n} \circ \eta_\infty$ for all n.

Ex. VII-8.3.(General category theory)

(a) Show quite generally from the arrow-theoretic definition that, given an infinite commutative ladder

$$\begin{array}{ccccccc} A_0 & \xleftarrow{\alpha_0} & A_1 & \xleftarrow{\alpha_1} & A_2 & \xleftarrow{\alpha_2} & \cdots \\ \zeta_0 \downarrow & & \zeta_1 \downarrow & & \zeta_2 \downarrow & & \\ B_0 & \xleftarrow{\beta_0} & B_1 & \xleftarrow{\beta_1} & B_2 & \xleftarrow{\beta_2} & \cdots \end{array}$$

(i.e. each square commutes) in which both lines have an inverse limit, there is a unique map

$$\zeta_\infty \ : \ \varprojlim(A_i, \alpha_i) \ \longrightarrow \ \varprojlim(B_i, \beta_i)$$

for which $\beta_{\infty n} \circ \zeta_\infty \ = \ \zeta_n \circ \alpha_{\infty n}$ for all n .

Here,

$$\varprojlim(A_i, \alpha_i) \ \text{ together with its maps } \ \alpha_{\infty n} \ : \ \varprojlim(A_i, \alpha_i) \ \to \ A_n$$

is any choice of an inverse limit (in the arrow-theoretic sense) for the top line in the display, and similarly for the bottom line.

(b) Furthermore, as long as ζ_n is an isomorphism for all sufficiently large n, then the map ζ_∞ is also an isomorphism (that is, a map with a 2-sided inverse with respect to composition). And so, *inverse limits are unique up to a unique isomorphism.*

Crucial Remark. We have such a commutative ladder

$$\begin{array}{ccccccccc} D_0 & \xleftarrow{\psi_0} & D_1 & \xleftarrow{\psi_1} & D_2 & \xleftarrow{\psi_2} & D_3 & \xleftarrow{\psi_3} & \cdots \\ \text{who cares?} \downarrow & & = \downarrow & & = \downarrow & & = \downarrow & & \\ \text{whatever} & \longleftarrow & [D_0 \rightharpoonup D_0] & \xleftarrow{\psi_1} & [D_1 \rightharpoonup D_1] & \xleftarrow{\psi_2} & [D_2 \rightharpoonup D_2] & \xleftarrow{\psi_3} & \cdots \end{array}$$

Now, D_∞, by **VII-8.2**, is the inverse limit of the top line, so, by **VII-8.3**(b), to show that $D_\infty \cong [D_\infty \rightharpoonup D_\infty]$, it remains only to prove the following :

Ex. VII-8.4. Show directly from the arrow-theoretic definition that

$[D_\infty \!\!\rightarrowtail\!\!\!\!\!\rightsquigarrow D_\infty]$ together with its maps $\eta_{\infty n} : [D_\infty \!\!\rightarrowtail\!\!\!\!\!\rightsquigarrow D_\infty] \to [D_n \!\!\rightarrowtail\!\!\!\!\!\rightsquigarrow D_n]$,

defined by $\eta_{\infty n}(f) = \theta_{\infty n} \circ f \circ \theta_{n\infty}$, is the inverse limit of the lower line,

$\varprojlim([D_i \!\!\rightarrowtail\!\!\!\!\!\rightsquigarrow D_i], \psi_{i+1})$, in the category of complete lattices and continuous maps.

Remark. The last three exercises complete the job, but the first and last may be a good challenge for most readers. They are likely to involve much of the material in the following more pedestrian approach to proving Scott's theorem (which is therefore not really a different proof), and which does exhibit the isomorphisms $D_\infty \overleftrightarrows [D_\infty \!\!\rightarrowtail\!\!\!\!\!\rightsquigarrow D_\infty]$ quite explicitly.

Ex. VII-8.5. (a) Show that $(\phi_0 \circ \psi_0)(f) \sqsubseteq f$ for all $f \in D_1$.
(b) Deduce that, for all n, we have $(\phi_n \circ \psi_n)(f) \sqsubseteq f$ for all $f \in D_{n+1}$.

The next exercise is best remembered as : "up, then down, (or, right, then left) always gives the identity map" whereas : "down, then up, (or left, then right) never gives a larger element" We're thinking of the objects as lined up in the usual way :

$$D_0 \qquad D_1 \qquad D_2 \qquad D_3 \qquad D_4 \quad \cdots\cdots \qquad D_\infty$$

Ex. VII-8.6. (a) Show that
(i) if $b \geq a$, then $\theta_{ba} \circ \theta_{ab} = \mathrm{id}_{D_a}$;
(ii) if $a \geq b$, then $(\theta_{ba} \circ \theta_{ab})(x) \sqsubseteq x$ for all x .
This, and the next part, include the cases when some subscripts are ∞ .

(b) More generally, deduce that, if $b < a$ and $b < c$, then

$$(\theta_{bc} \circ \theta_{ab})(x) \sqsubseteq \theta_{ac}(x) \quad \text{for all } x \in D_a \ ,$$

whereas, for all other a, b and c, the maps $\theta_{bc} \circ \theta_{ab}$ and θ_{ac} are actually equal.

Starting now we shall have many instances of $\bigsqcup_n d_n$. In every case, the sequence of elements d_n form a chain with respect to $\sqsubseteq$, and so a directed set, and thus the least upper bound does exist by completeness of D. Checking the 'chaininess' will be left to the reader.

Definitions. Define $\phi_\infty \ : \ D_\infty \ \to \ [D_\infty \!\rightharpoonup\! D_\infty]$ by

$$\phi_\infty(x) \ := \ \bigsqcup_k (\theta_{k,\infty} \circ (\theta_{\infty,k+1}(x)) \circ \theta_{\infty,k}) \ .$$

Define $\psi_\infty \ : \ [D_\infty \!\rightharpoonup\! D_\infty] \ \to \ D_\infty$ by

$$\psi_\infty(f) \ := \ \bigsqcup_n \theta_{n+1,\infty}(\theta_{\infty,n} \circ f \circ \theta_{n,\infty}) \ .$$

Ex. VII-8.7. Show that all the following are continuous :
(i) $\phi_\infty(x) : D_\infty \ \to \ D_\infty$ for all $x \in D_\infty$ (so ϕ_∞ is well-defined) ;
(ii) ϕ_∞ ; and
(iii) ψ_∞ .

Ex. VII-8.8. Prove that, for all $n, k < \infty$, and all $f \in D_{k+1}$, we have

$$\bigsqcup_n \theta_{n+1,\infty}(\theta_{k,n} \circ f \circ \theta_{n,k}) \ = \ \theta_{k+1,\infty}(f) \ ,$$

perhaps first checking that

$$\theta_{n+1,\infty}(\theta_{k,n} \circ f \circ \theta_{n,k}) \begin{cases} \sqsubseteq \ \theta_{k+1,\infty}(f) & \text{for } n \le k \ ; \\ = \ \theta_{k+1,\infty}(f) & \text{for } n \ge k \ . \end{cases}$$

(Oddly enough, this is not meaningful for n or k 'equal' to ∞ .)

Ex. VII-8.9. Show that

$$\theta_{\infty,k+1}(\bigsqcup_n \theta_{n+1,\infty}(z_{n+1})) \ = \ z_{k+1} \ ,$$

if $\{z_r \in D_r\}_{r>0}$ satisfies $\psi_r(z_{r+1}) = z_r$.

Ex. VII-8.10. Show that, for all $x \in D_\infty$,

$$\bigsqcup_k (\theta_{k+1,\infty} \circ \theta_{\infty,k+1})(x) \ = \ x \ .$$

Ex. VII-8.11. Show that, for all $f \in [D_\infty \!\rightharpoonup\! D_\infty]$,

$$\bigsqcup_k (\theta_{k,\infty} \circ \theta_{\infty,k} \circ f \circ \theta_{k,\infty} \circ \theta_{\infty,k}) \ = \ f \ .$$

Using these last four exercises, we can now complete the more mundane of the two proofs of Scott's theorem, by directly calculating that ϕ_∞ and ψ_∞ are mutually inverse :

For all $x \in D_\infty$,

$$\psi_\infty(\phi_\infty(x)) \;=\; \psi_\infty(\bigsqcup_k(\theta_{k,\infty} \circ (\theta_{\infty,k+1}(x)) \circ \theta_{\infty,k})) \qquad \text{(definition of } \phi_\infty)$$

$$= \bigsqcup_k \psi_\infty(\theta_{k,\infty} \circ (\theta_{\infty,k+1}(x)) \circ \theta_{\infty,k}) \qquad \text{(since } \psi_\infty \text{ is continuous)}$$

$$= \bigsqcup_k \bigsqcup_n \theta_{n+1,\infty}(\theta_{\infty,n} \circ \theta_{k,\infty} \circ (\theta_{\infty,k+1}(x)) \circ \theta_{\infty,k} \circ \theta_{n,\infty}) \qquad \text{(definition of } \psi_\infty)$$

$$= \bigsqcup_k \bigsqcup_n \theta_{n+1,\infty}(\theta_{k,n} \circ (\theta_{\infty,k+1}(x)) \circ \theta_{n,k}) \qquad \text{(by } \mathbf{VII-8.6}\text{(b))}$$

$$= \bigsqcup_k \theta_{k+1,\infty}(\theta_{\infty,k+1}(x)) \;=\; x\,, \quad \text{as required, by } \mathbf{VII-8.8} \text{ and } \mathbf{VII-8.10}.$$

For all $f \in [D_\infty \!\rightharpoonup\! D_\infty]$,

$$\phi_\infty(\psi_\infty(f)) \;=\; \bigsqcup_k(\theta_{k,\infty} \circ (\theta_{\infty,k+1}(\psi_\infty(f))) \circ \theta_{\infty,k}) \qquad \text{(definition of } \phi_\infty)$$

$$= \bigsqcup_k(\theta_{k,\infty} \circ (\theta_{\infty,k+1}(\bigsqcup_n \theta_{n+1,\infty}(\theta_{\infty,n} \circ f \circ \theta_{n,\infty}))) \circ \theta_{\infty,k}) \qquad \text{(definition of } \psi_\infty)$$

$$= \bigsqcup_k(\theta_{k,\infty} \circ (\theta_{\infty,k} \circ f \circ \theta_{k,\infty}) \circ \theta_{\infty,k}) \;=\; f \text{ as required, by } \mathbf{VII-8.11}.$$

The penultimate "=" uses **VII-8.9** with $z_{r+1} = \theta_{\infty,r} \circ f \circ \theta_{r,\infty}$; and that result is applicable because, rather trivially,

$$\psi_r(\theta_{\infty,r} \circ f \circ \theta_{r,\infty}) \;=\; \theta_{\infty,r-1} \circ f \circ \theta_{r-1,\infty}\ ,$$

employing the definition of ψ_r and **VII-8.6** .

The following fundamental result of Tarski about fixed points in complete lattices will be useful in the next subsection. Later we relate this to the fixedpoint combinator $\underline{Y}$ of Curry, called the *paradoxical combinator* by him.

Theorem VII-8.12. *Let D be a complete lattice, and consider operators $\Omega \in [D \multimap D]$. Define*

$$\mathsf{fix} \ : \ [D \multimap D] \ \to D \quad \textit{by} \quad \mathsf{fix}(\Omega) := \bigsqcup_n \Omega^n(\bot_D) \ .$$

Then $\mathsf{fix}(\Omega)$ is the minimal fixed point of Ω, and fix is itself continuous. In particular, any continuous operator on a complete lattice has at least one fixed point.

Proof. That fix is well-defined should be checked. As with all our other uses of $\bigsqcup_n$, the set of elements form a chain with respect to $\sqsubseteq$, and so a directed set, and the least upper bound does exist by completeness of D. To prove 'chaininess', we have

$$\bot_D \sqsubseteq \Omega(\bot_D) \ \text{ implies immediately that } \ \Omega^n(\bot_D) \sqsubseteq \Omega^{n+1}(\bot_D) \ .$$

As for the fixed point aspect, using continuity of Ω ,

$$\Omega(\mathsf{fix}(\Omega)) \ = \ \Omega(\bigsqcup_n \Omega^n(\bot_D)) \ = \ \bigsqcup_n \Omega^{n+1}(\bot_D) \ = \ \mathsf{fix}(\Omega) \ ,$$

as required, again using continuity of Ω .

As to the minimality, if $\Omega(d) = d$, then $\bot \sqsubseteq d$ gives $\Omega^n(\bot) \sqsubseteq \Omega^n(d) = d$ for all n, so

$$\mathsf{fix}(\Omega) \ = \ \bigsqcup_n \Omega^n(\bot_D) \ \sqsubseteq \ \bigsqcup_n \Omega^n(d) \ = \ \bigsqcup_n d \ = \ d \ ,$$

as required.

Finally, given a directed set $\mathbf{O} \subset [D \multimap D]$, we have

$$\mathsf{fix}(\bigsqcup_{\Omega \in \mathbf{O}} \Omega) \ = \ \bigsqcup_n (\bigsqcup_{\Omega \in \mathbf{O}} \Omega)^n(\bot) \ = \ \bigsqcup_n (\bigsqcup_{\Omega \in \mathbf{O}} \Omega^n)(\bot)$$

$$= \ \bigsqcup_n \bigsqcup_{\Omega \in \mathbf{O}} (\Omega^n(\bot)) \ = \ \bigsqcup_{\Omega \in \mathbf{O}} \bigsqcup_n (\Omega^n(\bot)) \ = \ \bigsqcup_{\Omega \in \mathbf{O}} \mathsf{fix}(\Omega) \ ,$$

as required, where justifications of the three middle equalities are left to the reader. And so, fix is continuous.

VII-9 Two (not entirely typical) Examples of Denotational Semantics.

We shall write out in all detail what presumably ought to be the denotational semantics of **ATEN** (earlier used to define computability), then illustrate it with a few examples. In the second half, we re-write the maps ρ_+ in the style of "denotational semantics for the λ-calculus", finishing with several interesting theorems about ρ_+ when the domain is D_∞, which also gives practice calculating in that domain.

Denotational semantics of ATEN.

Since *machine readability* and *executability* seem to be central concerns here, the formulas will all be given very technically, and we'll even begin with a super-formal statement of the syntax (though not exactly in the standard BNF-style).

Here it all is, in one largely human-unreadable page. See the following remarks.

$BRA := \{\,)\,,\,(\,\}$,which merely says that we'll have lots of brackets, despite some CSers' abhorrence.

$v \in IDE := \{\, x \mid * \,||\, x \,||\, v* \,\}$ which says, e.g. that the 'real' x_3 is $x***$, and x_0 is just x.

$$s,t \in EXP := \{\, BRA \mid IDE \mid + \mid \times \mid 0 \mid 1 \,||\, v \mid 0 \mid 1 \,||\, (s+t) \mid (s \times t) \,\}\,.$$

$$F,G \in EXP' := \{\, BRA \mid EXP \mid < \mid \approx \mid \neg \mid \wedge \,||\, s < t \mid s \approx t \,||\, \neg F \mid (F \wedge G) \,\}\,.$$

$$C,D \in COM := \{\, BRA \mid IDE \mid EXP \mid EXP' \mid \twoheadleftarrow \mid ; \mid \mathbf{whdo} \,||\, v \twoheadleftarrow t \,||\, (C;D) \mid \mathbf{whdo}(F)(C) \,\}\,.$$

$$VAL \text{ “}=\text{”} \{\perp_{\mathbf{N}}\} \cup \mathbf{N} \quad ; \quad BOOL \text{ “}=\text{”} \{\, \perp_{BOOL}\,,\, \mathsf{Tr}\,,\, \mathsf{Fs}\,\} \quad ; \quad STATE = [IDE \rightharpoonup VAL]\,.$$

$$\mathcal{E} : EXP \to [STATE \rightharpoonup VAL] \qquad \text{defined by}$$

$$\mathcal{E}[\![v]\!](\sigma) := \sigma(v) \quad ; \quad \mathcal{E}[\![0]\!](\sigma) := 0_{\mathbf{N}} \quad ; \quad \mathcal{E}[\![1]\!](\sigma) := 1_{\mathbf{N}} \quad ;$$

$$\mathcal{E}[\![(s+t)]\!](\sigma) := \mathcal{E}[\![s]\!](\sigma) +_{\mathbf{N}} \mathcal{E}[\![t]\!](\sigma) \quad ; \quad \mathcal{E}[\![(s \times t)]\!](\sigma) := \mathcal{E}[\![s]\!](\sigma) \cdot_{\mathbf{N}} \mathcal{E}[\![t]\!](\sigma) \quad .$$

(Note that $\perp_{\mathbf{N}}$ is produced when it is either of the inputs for $+_{\mathbf{N}}$ or for $\cdot_{\mathbf{N}}$.)

$$\mathcal{E}' \ : \ EXP' \ \rightarrow \ [STATE \rightharpoonup BOOL] \qquad \text{defined by}$$

saying firstly that $\mathcal{E}'[\![F]\!](\sigma) := \perp_{BOOL}$ if either F is an atomic formula involving a term s with $\mathcal{E}[\![s]\!](\sigma) := \perp_{\mathbf{N}}$, or if F is built using $\neg$ or $\wedge$ using a formula G for which $\mathcal{E}'[\![G]\!](\sigma) := \perp_{BOOL}$; otherwise

$$\mathcal{E}'[\![s < t]\!](\sigma) := \text{if } \mathcal{E}[\![s]\!](\sigma) <_{\mathbf{N}} \mathcal{E}[\![t]\!](\sigma), \text{ then } \mathsf{Tr}, \text{ else } \mathsf{Fs} \ ;$$

$$\mathcal{E}'[\![s \approx t]\!](\sigma) := \text{if } \mathcal{E}[\![s]\!](\sigma) = \mathcal{E}[\![t]\!](\sigma), \text{ then } \mathsf{Tr}, \text{ else } \mathsf{Fs} \ ;$$

$$\mathcal{E}'[\![\neg F]\!](\sigma) := \text{if } \mathcal{E}'[\![F]\!](\sigma) = \mathsf{Fs}, \text{ then } \mathsf{Tr}, \text{ else } \mathsf{Fs} \ ;$$

$$\mathcal{E}'[\![(F \wedge G)]\!](\sigma) := \text{if } \mathcal{E}'[\![F]\!](\sigma) = \mathsf{Tr} \text{ and } \mathcal{E}'[\![G]\!](\sigma) = \mathsf{Tr}, \text{ then } \mathsf{Tr}, \text{ else } \mathsf{Fs} \ .$$

$$\mathcal{C} \ : \ COM \ \rightarrow \ [STATE \rightharpoonup STATE] \qquad \text{defined by}$$

$$\mathcal{C}[\![v \Leftarrow t \ ]\!](\sigma) := \sigma^{[v \mapsto \mathcal{E}[\![t]\!](\sigma)]} \quad \text{where } \sigma^{[v \mapsto \alpha]} \text{ agrees with } \sigma, \text{ except } v \mapsto \alpha \ ;$$

$$\mathcal{C}[\![(C; D)]\!] := \mathcal{C}[\![D]\!] \circ \mathcal{C}[\![C]\!] \ ;$$

$$\mathcal{C}[\![\mathbf{whdo}(F)(C)]\!] := \mathsf{fix}(f \mapsto (\sigma \mapsto (\text{if } \mathcal{E}'[\![F]\!](\sigma) = \mathsf{Tr}, \text{ then } (f \circ \mathcal{C}[\![C]\!])(\sigma), \text{ else } \sigma \ ))) \ ,$$

where the fixpoint operator, $\mathsf{fix} \ : \ [D \rightharpoonup D] \ \rightarrow D$, has $D \ = \ [STATE \rightharpoonup STATE]$.

Remarks. The first five displays are the syntax. All but the first give a set of strings in the usual three stages: { first, all symbols to be used || next, which are the atomic strings || and finally, how the 'production' of new strings is done (induction on structure)}. To the far left are 'typical' member(s) of the string set to be defined, those being then used in the production and also on lines further down. [Where whole sets of strings are listed on the left of the symbols-to-be-used listing and other times as well, a human interpreter thinks of those substrings as single symbols, when they appear inside the strings being specified on that line. For example, $(x***+x**)$ has ten symbols, but a human ignores the brackets and thinks of three, i.e. $x_3 + x_2$—and maybe even as a single symbol, since, stretching it a bit, however complicated, a term occurring inside a formula is intuited as a single symbol in a sense, as a component of the formula. And similarly, a term or a formula within a command is psychologically a single symbol. Sometimes the phrase "immediate constituents" is used for this, in explaining below the inductive nature of the semantic function definitions.]

The last of the five lines is the syntax of **ATEN** itself. The third and fourth lines give the syntax of the assertion language, first terms, then (quantifier-free) formulas, in $1^{\underline{\text{st}}}$order number theory. We've used names

which should remind CSers of the word "expression", but they might better be called $TRM(= EXP)$ and $FRM_{\text{free}}(= EXP')$ from much earlier stuff here.

Often, CSers would lump EXP and EXP' together into a single syntactic category. Why they do so is not unmysterious to me; I am probably missing some considerable subtlety. But I am well aware of the confusion that tendency in earlier CS courses causes for average students when I teach them logic. They initially find it strange that I should make a distinction between strings in a formal language which we intuit as standing for objects, and other strings which we intuit as standing for statements! On the other hand, I myself find it strange to think of $(3 < 4) + ((5 = 2) + 6)$ as any kind of expression! (and not because $5 \neq 2$) Another inscrutability for me is that many of the technicalities on the previous page would normally be written down in exactly the opposite order in a text on programming languages! While I'm in the mood to confess a mentality completely out of synch with the CSers, here's another bafflement which some charitable CSer will hopefully straighten me out on. Within programs in, say, PASCAL, one sees the *begin—end* and use of indentation as a replacement for brackets, which makes things much more readable (if *any* software could ever be described as (humanly) readable). But there seems to be a great desire to avoid brackets, and so avoid non-ambiguity, in writing out the syntax of languages. Instead, vague reference is made to parsers. I can appreciate a desire sometimes to leave open different possibilities. A good example is the λ-calculus, where using $A(B)$ rather than (AB) in the basic syntactic setup is a possibility, and probably better for psychological purposes, but not for economy. And the semantics is largely independent of the parsing. But when precision is of utmost importance, I cannot understand this tendency to leave things vague. Of course, there still remains a need to specify algorithms for deciding whether a string (employing the basic symbols of the language) is actually in the language. But surely that's a separate issue.

The sixth display gives information about the semantic domains needed. We haven't said what VAL actually "is" (hence the "="), other than that it contains as elements $\perp$ and all the natural numbers. Below we come clean on that. The use of $[\ \ \multimap\ \ ]$ is some indication that these semantic sets have some structure, actually a partial order.

Finally, the three semantic functions, corresponding to terms, formulae, and commands, are given. We have been super-strict in distinguishing notationally between the symbol "+" and the actual operation "$+_{\mathbf{N}}$" on natural numbers, and similarly for "$\cdot_{\mathbf{N}}$" and "$<_{\mathbf{N}}$". Some elementary confusions revolve around this point, in the presence of ambiguous notation, which point otherwise might seem overly fussy. We have, in the same vein, used our usual "$\approx$" as the formal equality symbol, to distinguish it from the actual relation of sameness. Each of the three semantic functions is defined by structural induction on the productions defined at the right-hand ends of the corre-

sponding syntactic sets. As mentioned in defining $\mathcal{E}$, the operations $+_{\mathbf{N}}$ and $\cdot_{\mathbf{N}}$ produce $\perp_{\mathbf{N}}$ if either or both of their inputs is $\perp_{\mathbf{N}}$. And from the definition of $\mathcal{E}'$, the relation $<_{\mathbf{N}}$ has no need to concern itself with comparing $\perp_{\mathbf{N}}$ with anything.

As for $\mathcal{E}$, we're just saying how, using $\mathcal{E}$'s adjoint, a term together with a state will yield a number, exactly as in Tarski's definition of truth. In fact,

$\mathcal{E}[\![t]\!](\sigma)$ is just another name for $t^{\underline{v}}$, where $\underline{v}$ is identified with that state σ mapping each x_n to v_n.

See [**LM**], beginning of Ch.6.

As for $\mathcal{E}'$, we're just saying how, after taking its adjoint, a formula together with a state will yield a truth value, exactly as in Tarski's definition of truth. In fact,

Tr and Fs for $\mathcal{E}'[\![F]\!](\sigma)$ are just other ways of saying whether $\underline{v} \in F^V$ or not.

See [**LM**] for F^V. Alternatively, $\underline{v} \in F^V$ says "F is true at $\underline{v}$ from V".

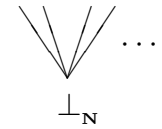
As for $\mathcal{C}$, we're just saying how, after taking its adjoint, a command together with a state will yield another state, exactly as in our original definition of the semantics of **BTEN**. In fact,

$\mathcal{C}[\![C]\!](\sigma)$ is just another name for $||C||(\underline{v})$; i.e. $\mathcal{C}[\![C]\!]$ is another name for $||C||$.

The right-hand sides in the definitions of $\mathcal{E}'$ and $\mathcal{C}$ use an (if-then-else) in the spirit of **McSelf** and should be taken that way. (See also the discussion just below the diagram in the next paragraph.) In [**Al**] Ch.5, a program in PASCAL is written out soon after specifying the denotational semantics of his small language. Presumably LISP would be just as good. Such a program is called an *implementer*. Another one of my confusions is the question of why implementing a simple language inside a very complicated practical language is *useful*. (I can see where it would be *fun*.) Perhaps it is related to the fact that the implementer is just a *single* program in that complex language, a single program in which one can develop considerable confidence. I am certainly not (yet?) the one to write an implementation of **ATEN** in any real programming language.

Note that for the final clause, giving the denotation of the **whdo**-command, we need the discussion of fixed points from the end of the previous subsection. This specification of the **whdo**-command comes from the remarks at the end of Subsection VII-7, and is evidently far more 'implementable' than

that in the originally specified semantics of **BTEN**. If not for that one command, this whole exercise would presumably be regarded as rather sterile. Indeed, as mentioned earlier, it's when applied to much more complicated (e.g. ALGOL-like) languages that this impression of sterility dissipates. The need for anything like a self-reflective domain as constructed in the previous subsection is unclear. But we do at least need that $D = [STATE \mathrel{-\!\!\!\rightharpoonup} STATE]$ is a lattice for which Tarski's theorem on minimum fixpoint operators works. That follows as long as $STATE$ is a complete lattice, which itself follows as long as VAL is. Therefore we complete the unfinished business on the previous page by specifying

VAL := the flat lattice 

and $BOOL$ similarly.

There are a couple of points worth adding, to shore up the semantic definitions. Suppose we were dealing with **BTEN** rather than **ATEN**, so we had to give the semantics of the command $\mathbf{ite}(F)(C)(D)$, that is, the (*if-then-else*)-command. In the style previous this would be

$$\mathcal{C}[\![\mathbf{ite}(F)(C)(D)]\!](\sigma) \;:=\; \text{if } \mathcal{E}'[\![F]\!](\sigma) = \mathsf{Tr},\ \text{then } \mathcal{C}[\![C]\!](\sigma),\ \text{else } \mathcal{C}[\![D]\!](\sigma)\ .$$

To my complete bafflement, [**Al**], top of p.12 would object strenuously to this, insisting that it must be

$$\mathcal{C}[\![\mathbf{ite}(F)(C)(D)]\!](\sigma) \;:=\; (\text{if } \mathcal{E}'[\![F]\!](\sigma) = \mathsf{Tr},\ \text{then } \mathcal{C}[\![C]\!],\ \text{else } \mathcal{C}[\![D]\!])(\sigma)\ .$$

Once again, I am in need of a kind CSer to straighten me out on a subtle point. Furthermore, the definition in [**Go**], p.51 (C3) , seems not to be consistent in terms of the domains involved with the earlier definitions there, though that seems to be fixable by extending the cond-function there in a certain way. But perhaps again I need a tutorial. In any case here is a re-written version of our definition more in that style, which is similar to [**St**], p.196. (He has the added complication of "side-effects" to deal with, but in their absence, his definition reduces to exactly the following, at least up to currying/uncurrying.) Further down we do the same for the **whdo**-command, to which similar bafflements on my part apply with respect to the

versions in [**Al**] and [**Go**]. Re-define (without really changing the definition)

$$\mathcal{C}[\![\mathbf{ite}(F)(C)(D)]\!] \;:=\; con \circ (\mathcal{E}'[\![F]\!] \times \mathcal{C}[\![C]\!] \times \mathcal{C}[\![D]\!]) \circ ddg \;,$$

where

$$ddg : STATE \to STATE \times STATE \times STATE \quad ; \quad \sigma \mapsto (\sigma, \sigma, \sigma)$$

is the double diagonal map, and

$$con : BOOL \times STATE \times STATE \to STATE$$

$$(Tr, \sigma, \tau) \mapsto \sigma \quad ; \quad (Fs, \sigma, \tau) \mapsto \tau \quad ; \quad (\perp_{BOOL}, \sigma, \tau) \mapsto \quad \perp_{STATE} \quad ;$$

is the conditional. (This has turned out a bit simpler than in the above references, with no need to fool around with defining a $\star$-operator, partly because we are not insisting on currying everything, with its attendant contortions.)

In any case, this isolates the facts that we definitely need *ddg* and *con* to be continuous, which they are, but otherwise no further comment is needed.

Now we can re-write the *while-do* semantic definition also in this style :

$$\mathcal{C}[\![\mathbf{whdo}(F)(C)]\!] \;:=\; fix\ [\ f \;\mapsto\; con \circ (\mathcal{E}'[\![F]\!] \times (f \circ \mathcal{C}[\![C]\!]) \times Id) \circ ddg\ ]\ .$$

Some examples of the definition.

The first three examples below are utterly simple programs in **ATEN**, to the point of being silly. And the purpose of denotational semantics, though not crystal clear to me, is certainly *not* to be able to write out particular examples. But the ones below should serve to better familiarize us with how the definition works, and, more importantly, to give some confidence that the answers are coming out 'right'.

(1) If $C \;=\; \mathbf{whdo}(x_0 \approx x_0)(D)$ for some command D, we obviously realize an infinite loop no matter what is in the bins to start, i.e. for any initial state. Let's figure out $\mathcal{C}[\![C]\!]$. Using the definition, we get

$$\mathsf{fix}(f \mapsto (\sigma \mapsto (f \circ \mathcal{C}[\![D]\!])(\sigma))) \;=\; \mathsf{fix}(f \mapsto f \circ \mathcal{C}[\![D]\!])\ ,$$

since certainly $\mathcal{E}'[\![x_0 \approx x_0]\!](\sigma) = Tr$ for all σ . Now define a function f_0 in $[STATE \multimap STATE]$ by $f_0(\sigma) = \perp$ for all σ . It is very clear that

$f_0 \sqsubseteq f$ for all $f \in [STATE \text{—}\!\rightsquigarrow STATE]$. Also $f_0 \circ g = f_0$ for any function g in $[STATE \text{—}\!\rightsquigarrow STATE]$, since f_0 is a constant function. So f_0 is the required answer, that is, the minimal fixed point of the operator which is 'composition on the left with $\mathcal{C}[\![D]\!]$'. And surely f_0 *should be* the element in the domain $[STATE \text{—}\!\rightsquigarrow STATE]$ which denotes a program that always loops ! A better name for that function is $\perp_{[STATE \rightsquigarrow STATE]}$. The previous $\perp$ is $\perp_{STATE} = \perp_{[IDE \rightsquigarrow VAL]}$, namely the function which maps all $v \in IDE$ to $\perp_{VAL}$, which Dana Scott refers to as "the undefined".

(2) If $C = \mathbf{whdo}(x_0 < 1)(x_0 \leftarrow\!\!\div\, x_0 + 1)$, we see that any initial state is unchanged by the command, except when bin 0 contains zero. In the latter case, the zero in bin 0 is changed to 1, and then the process terminates. First here are three preliminary calculations:

$$\mathcal{E}[\![x_0 + 1]\!](\sigma) = \mathcal{E}[\![x_0]\!](\sigma) +_{\mathbf{N}} \mathcal{E}[\![1]\!](\sigma) = \sigma(x_0) +_{\mathbf{N}} 1_{\mathbf{N}} \ .$$

$$\mathcal{C}[\![x_0 \leftarrow\!\!\div\, x_0 + 1]\!](\sigma) = \sigma^{[x_0 \,\mapsto\, \mathcal{E}[\![x_0+1]\!](\sigma)]} = \sigma^{[x_0 \,\mapsto\, 1_{\mathbf{N}} +_{\mathbf{N}} \sigma(x_0)]} \ .$$

$$\mathcal{E}'[\![x_0 < 1]\!] \ = \ \begin{cases} Tr & \text{if } \sigma(x_0) <_{\mathbf{N}} 1_{\mathbf{N}} \ ; \\ Fs & \text{otherwise .} \end{cases}$$

Let's figure out $\mathcal{C}[\![C]\!]$. Using the definition, we get

$$\mathsf{fix}(f \mapsto (\sigma \mapsto (\text{if } \mathcal{E}'[\![x_0 < 1]\!](\sigma) = \mathsf{Tr}, \text{then } (f \circ \mathcal{C}[\![x_0 \leftarrow\!\!\div\, x_0 + 1]\!])(\sigma), \text{else } \sigma\ )))$$

$$= \ \mathsf{fix}(f \mapsto (\sigma \mapsto \begin{cases} f(\sigma^{[x_0 \,\mapsto\, 1_{\mathbf{N}}]}) & \text{if } \sigma(x_0) = 0_{\mathbf{N}} \ ; \\ \sigma & \text{if } \sigma(x_0) \neq 0_{\mathbf{N}} \ . \end{cases}$$

Now let f_1 be *any* fixed point of the latter operator. So

$$f_1(\sigma) = \begin{cases} f_1(\sigma^{[x_0 \,\mapsto\, 1_{\mathbf{N}}]}) & \text{if } \sigma(x_0) = 0_{\mathbf{N}} \ ; \\ \sigma & \text{if } \sigma(x_0) \neq 0_{\mathbf{N}} \ . \end{cases}$$

But the lower line then determines the upper line, and we conclude that there is only one fixed point in this case (no minimization needed!), given by

$$f_1(\sigma) = \begin{cases} \sigma^{[x_0 \,\mapsto\, 1_{\mathbf{N}}]} & \text{if } \sigma(x_0) = 0_{\mathbf{N}} \ ; \\ \sigma & \text{if } \sigma(x_0) \neq 0_{\mathbf{N}} \ . \end{cases}$$

Well, this last function is exactly the one the command was supposed to compute, i.e. 'if necessary, change the 0 in bin zero to a 1, and do nothing else', so our definition is doing the expected here as well.

(3) If $C = \mathbf{whdo}(x_0 < 1)(x_1 \div x_1)$, we see that any initial state is unchanged by the command, except when bin zero contains 0. In the latter case, the program does an infinite loop.
First here are the three preliminary 'calculations':

$$\mathcal{E}[\![x_1]\!](\sigma) \;=\; \sigma(x_1) \;;$$

$$\mathcal{C}[\![x_1 \div x_1]\!](\sigma) \;=\; \sigma^{[x_1 \,\mapsto\, \mathcal{E}[\![x_1]\!](\sigma)]} \;=\; \sigma^{[x_1 \,\mapsto\, \sigma(x_1)]} = \sigma \;,$$

hardly surprising; and, as before

$$\mathcal{E}'[\![x_0 < 1]\!] \;=\; \begin{cases} Tr & \text{if } \sigma(x_0) <_{\mathbf{N}} 1_{\mathbf{N}} \;; \\ Fs & \text{otherwise .} \end{cases}$$

From the definition, $\mathcal{C}[\![C]\!]$ is given by

$$\mathsf{fix}(f \mapsto (\sigma \mapsto (\text{if } \mathcal{E}'[\![x_0 < 1]\!](\sigma) = \mathsf{Tr}, \text{then } (f \circ \mathcal{C}[\![x_1 \div x_1]\!])(\sigma), \text{else } \sigma\ )))$$

$$= \mathsf{fix}(f \mapsto (\sigma \mapsto \begin{cases} f(\sigma) & \text{if } \sigma(x_0) = 0_{\mathbf{N}} \;; \\ \sigma & \text{if } \sigma(x_0) \neq 0_{\mathbf{N}} \;. \end{cases}$$

Suppose that f_2 is a fixed point of the latter operator. So

$$f_2(\sigma) \;=\; \begin{cases} f_2(\sigma) & \text{if } \sigma(x_0) = 0_{\mathbf{N}} \;; \\ \sigma & \text{if } \sigma(x_0) \neq 0_{\mathbf{N}} \;. \end{cases}$$

The top line says nothing, and any such function *is* a fixed point. Thus, clearly f_3 is the minimal fixed point, where

$$f_3(\sigma) \;=\; \begin{cases} \perp & \text{if } \sigma(x_0) = 0_{\mathbf{N}} \;; \\ \sigma & \text{if } \sigma(x_0) \neq 0_{\mathbf{N}} \;. \end{cases}$$

Once again, this last function is exactly the one the command was supposed to compute, i.e. 'loop if bin 0 has a 0, otherwise, terminate after doing nothing'. So our definition is doing the expected, giving at least a little reinforcement to our confidence in the technicalities.

(4) Now we give a more extended (and possibly interesting) example, using the obvious non-recursive algorithm that calculates the factorial function. In [**Al**], p.55, this is also an example, with very abbreviated explanation, so the reader may be interested to compare. A major contributor to the much

more detail here (besides kindness to the reader) is the fact that **ATEN** is so primitive. The simple language in [**Al**] at least has subtraction and negative integers, which **ATEN** doesn't. But the details below are probably instructive, even in the direction of writing an general algorithm to translate programs in that language into commands in **ATEN**.

We would like to use the 'command'

$$x_0 \leftarrow\!\!\div\ 1\ ;\ \mathbf{whdo}(0 < x_1)(x_0 \leftarrow\!\!\div\ x_1 \times x_0\ ;\ \text{“}x_1 \leftarrow\!\!\div\ x_1 - 1\text{”})\ .$$

After approximately "v_1" cycles, this should terminate, leaving the number "v_1!" in bin zero (where the natural number v_i is the initial content of bin number i, i.e. it is $\sigma(x_i)$ below). And below we demonstrate that the semantic definitions prove this fact. The quotation marks are there because $x_1 - 1$ is not a term. We take "$x_1 \leftarrow\!\!\div\ x_1 - 1$" to be an abbreviation for

$$x_2 \leftarrow\!\!\div\ 0\ ;\ \mathbf{whdo}(\neg x_2 + 1 \approx x_1)(x_2 \leftarrow\!\!\div\ x_2 + 1)\ ;\ x_1 \leftarrow\!\!\div\ x_2\ ,$$

omitting associativity brackets for ";". So this produces that predecessor function which is undefined on 0 (but the function's value ends up in bin1, not bin 0).

So we've got a fairly lengthy job, much of which will be left as exercises for the reader.

First show that

$$\mathcal{C}[\![\mathbf{whdo}(\neg x_2{+}1 \approx x_1)(x_2 \leftarrow\!\!\div x_2{+}1)]\!] = \mathsf{fix}(f \mapsto (\sigma \mapsto \begin{cases} f(\sigma^{[x_2 \mapsto \sigma(x_2)+1]}) & \text{if } \sigma(x_2)+1 \neq \sigma(x_1)\ ; \\ \sigma & \text{if } \sigma(x_2)+1 = \sigma(x_1)\ . \end{cases}$$

Then argue (by induction on $\sigma(x_1) - \sigma(x_2)$ in the first case) that the minimal fixed point here is given by

$$\sigma \mapsto \begin{cases} \sigma^{[x_2 \mapsto \sigma(x_1)-1]} & \text{if } \sigma(x_2) < \sigma(x_1)\ ; \\ \bot & \text{if } \sigma(x_2) \geq \sigma(x_1)\ . \end{cases}$$

Now argue that

$$\mathcal{C}[\![\text{“}x_1 \leftarrow\!\!\div\ x_1 - 1\text{”}]\!] \quad = \quad \sigma \mapsto \begin{cases} \sigma^{[x_2 \mapsto \sigma(x_1)-1][x_1 \mapsto \sigma(x_1)-1]} & \text{if } \sigma(x_1) \neq 0\ ; \\ \bot & \text{if } \sigma(x_1) = 0\ . \end{cases}$$

Next argue that, if $E = (x_0 \leftarrow\!\!\div\ x_1 \times x_0\ ;\ \text{“}x_1 \leftarrow\!\!\div\ x_1 - 1\text{”})$, then

$$\mathcal{C}[\![E]\!] \quad = \quad \sigma \mapsto \begin{cases} \sigma^{[x_0 \mapsto \sigma(x_1)\sigma(x_2)][x_1 \mapsto \sigma(x_1)-1][x_2 \mapsto \sigma(x_1)-1]} & \text{if } \sigma(x_1) \neq 0\ ; \\ \bot & \text{if } \sigma(x_1) = 0\ . \end{cases}$$

The next step is yet another application of the definition of the semantic function on **whdo**-commands to yield

$$\mathcal{C}[\![\mathbf{whdo}(0 < x_1)(E)]\!] = \mathsf{fix}(f \mapsto (\sigma \mapsto \begin{cases} \sigma^{[x_0 \mapsto \sigma(x_1)\sigma(x_2)][x_1 \mapsto \sigma(x_1)-1][x_2 \mapsto \sigma(x_1)-1]} & \text{if } \sigma(x_1) \neq 0 \ ; \\ \sigma & \text{if } \sigma(x_1) = 0 \ . \end{cases}$$

A more subtle argument than the earlier analogues then yields the minimal fixed point as

$$\sigma \mapsto \begin{cases} \sigma^{[x_0 \mapsto \sigma(x_1)!\sigma(x_0)][x_1 \mapsto 0][x_2 \mapsto 0]} & \text{if } \sigma(x_1) \neq 0 \ ; \\ \sigma & \text{if } \sigma(x_1) = 0 \ . \end{cases}$$

Finally , we can easily see that

$$\mathcal{C}[\![x_0 \Leftarrow 1 \ ; \ \mathbf{whdo}(0 < x_1)(E)]\!] \ = \ \sigma \mapsto \begin{cases} \sigma^{[x_0 \mapsto \sigma(x_1)!][x_1 \mapsto 0][x_2 \mapsto 0]} & \text{if } \sigma(x_1) \neq 0 \ ; \\ \sigma^{[x_0 \mapsto 1]} & \text{if } \sigma(x_1) = 0 \ . \end{cases}$$

This is the required result—our original command computes a function which puts "$\sigma(x_1)!$" into bin zero (to mix up the two ways of thinking about states), and which does irrelevant things to bins 1 and 2.

Denotational semantics of Λ

This consists of super-formal statements of the syntax from the very beginning, about 100 pages back, and of the specifications in the definition of *λ-model*. See the first half of this subsection for an analogue and an explanation of the syntax specification in the first three lines below. Here it is, again largely human-unreadable, but perhaps more mechanically translatable for writing an implementation. See Ch. 8 of **[St]** for much more on this.

$$BRA \ := \ \{ \ ) \ , \ (\ \} \quad .$$

$$v \in IDE \ := \ \{ \ x \mid \ * \ || \ x \ || \ v * \ \} \quad .$$

$$A, B \in \Lambda = EXP \ := \ \{ \ BRA \mid IDE \mid \lambda \mid \ \bullet \ || \ v \ || \ (AB) \mid (\lambda v \bullet A) \ \} \ .$$

$$VAL \ = \ \text{any } \lambda\text{−model} \ ,$$

which therefore includes a binary operation called $\cdot$, and a function ψ.

$$ENV \ = \ [IDE \to VAL] \ .$$

For $\rho \in ENV$, $v \in IDE$, and $d \in VAL$, define $\rho^{[v \mapsto d]} \in ENV$ to agree with ρ except it maps the variable ('identifier') v to d .

$$\mathcal{E} \ : \ EXP \ \to \ [ENV \to VAL] \qquad \text{is defined by}$$
$$\mathcal{E}[\![v]\!](\rho) := \rho(v) \ \ ;$$
$$\mathcal{E}[\![(AB)]\!](\rho) := \mathcal{E}[\![A]\!](\rho) \cdot \mathcal{E}[\![B]\!](\rho) \quad ;$$
$$\mathcal{E}[\![(\lambda v \bullet A)]\!](\rho) := \psi(d \ \mapsto \ \mathcal{E}[\![A]\!](\rho^{[v \mapsto d]}) \quad .$$

Note that the last three lines are just the three basic properties, in a messier notation, describing how what we called ρ_+ behaves in a λ-model. Note also that the last line is only meaningful in this structural inductive definition with an accompanying inductive proof that the map to which we are applying ψ is in fact in $[VAL \rightsquigarrow VAL]$.

The other two fundamental proofs wanted here are of the facts that

$$(*) \qquad A \ \approx \ B \qquad \Longrightarrow \qquad \mathcal{E}[\![A]\!] \ = \ \mathcal{E}[\![B]\!] \ ; \ \text{i.e.} \ \ \rho_+(A) \ = \ \rho_+(B) \ \ \forall \rho \ \ ;$$

and that

$$(**) \quad \mathcal{E}[\![A^{[x\mapsto B]}]\!](\rho) \;=\; \mathcal{E}[\![A]\!](\rho^{[x\mapsto \mathcal{E}[\![B]\!](\rho)]}) \;;\; \text{i.e. } \rho_+(A^{[x\mapsto B]}) \;=\; \rho_+^{[x\mapsto \rho_+(B)]}(A) \;.$$

None of these proofs requires any great cleverness.

Later we discuss how, after passing to equivalence classes, and when Scott's D_∞ is VAL, the map on $EXP/\approx$ coming from ρ_+ is not injective, but that it becomes so when restricted to the set of equivalence classes of closed terms which have a normal form; i.e. closed normal form terms all map under this denotational semantics to different objects in the λ-model, this being independent of ρ.

We'll now go back to the handier ρ_+ notation.

Proving the theorem of David Park just below gives an opportunity for illustrations of this "denotational semantics of the λ-calculus" (and especially for calculations to increase one's familiarity with the innards of Scott's D_∞).

Recall that Curry's fixpoint (or "paradoxical") combinator $\underline{Y}$ is a closed term, so it is unambiguous to define

$$Y \;:=\; \rho_+(\underline{Y}) \;\in\; D_\infty \quad ,$$

i.e. it is independent of the particular ρ used.

Theorem VII-9.1. *Under the canonical isomorphism*

$$[[D_\infty \!\rightharpoonup\! D_\infty] \!\rightharpoonup\! D_\infty] \quad \overset{\cong}{\longleftrightarrow} \quad D_\infty \;,$$

the Tarski fixpoint operator, fix *(from* **VII-8.12***), corresponds to* Y*, the image of Curry's fixpoint combinator. Three equivalent expressions of this are*

$$\phi_\infty(Y) = \mathsf{fix} \circ \phi_\infty \quad ; \quad Y = \psi_\infty(\mathsf{fix} \circ \phi_\infty) \quad ; \quad \mathsf{fix} = \phi_\infty(Y) \circ \psi_\infty \;.$$

We shall give the proof as a sequence of exercises, ending with establishing the first of those three. The fact that the three are equivalent is staggeringly trivial, via $\psi_\infty \;=\; \phi_\infty^{-1}$. The only two proofs of the theorem that I have seen are the scanned image of Park's succinct typescript from 1976-78, and the proof in **[Wa]** (which we give, with proofs of basic identities he uses).

This proof depends on the explicit ϕ_0 and ψ_0 (but not any explicit D_0) from Scott's building of D_∞ from D_0, as exposited here in VII-8. Park sketches how a different choice of (ϕ_0, ψ_0) gives a different answer. So this theorem definitely does not generalize as is, to an arbitrary extensional λ-model with compatible complete lattice structure. Later we use Park's theorem a couple of times to give other explicit results about the denotational semantics of Λ, including another fixpoint operator, though not Turing's fixpoint combinator (which the reader might like to think about).

First here are some useful exercises and notation.

Definition. Abbreviate $\theta_{n,\infty} \circ \theta_{\infty,n}$ to just $\pi_n : D_\infty \to D_\infty$.

Then π_n is a projection operator (i.e. $\pi_n \circ \pi_n = \pi_n$), whose image is a sublattice of D_∞ which is isomorphic to D_n, and often identified with D_n .

Much of the literature makes the notation even more succinct, denoting $\pi_n x$ as x_n . This can be very efficient, but a bit confusing for us beginners. Firstly, it's rather more natural in mathematics to use x_n as the nth component of x, that is, to write x as $(x_1, x_2, x_3, \cdots)$. So this latter x_n would be in D_n, not in D_∞ . To avoid conflict with the literature, we won't use x_n at all. Another source of confusion with this subscripting is that an expression such as $y_{n+1}(x_n)$ sits in one's consciousness more naturally as an element of $D_{n+1} = [D_n \multimap D_n]$ acting on an element of D_n to produce an element of D_n . But in fact it must (and does in the literature) mean what in the above defined notation is $\phi_\infty(\pi_{n+1}y)(\pi_n x)$. (The two confusable meanings are of course closely related—see for example **Ex. 8** below.) This last expression is becoming unwieldy, and we'll at least be able to simplify it to $\pi_{n+1}y \cdot \pi_n x$.

Definition. Let "$\cdot$" be the binary operation on D_∞ adjoint to ϕ_∞ ; that is,

$$a \cdot b := \phi_\infty(a)(b) .$$

This is the binary operation from earlier in the general theory of λ-models.

(Perhaps annoying the experts) for a bit more clarity we continue with juxtaposition for the operation in Λ, but use "$\cdot$" in D_∞ . Thus, a basic

property of the maps from denotational semantics is

$$\rho_+(\underline{A}\ \underline{B}) \ = \ \rho_+(\underline{A}) \cdot \rho_+(\underline{B}) \qquad \text{for all } \underline{A} \text{ and } \underline{B} \text{ in } \Lambda \ .$$

Readers who wish to go further with this and consult the literature will need to accustom themselves to two 'identifications' which we are avoiding for conceptual clarity (but giving ourselves messier notation) :

(i) identifying Λ with its image $\rho_+(\Lambda) \ \subset \ D_\infty$, which *is* ambiguous on *non-closed* terms in Λ , and also ambiguous in another sense 'up to $\approx$' ;

(ii) identifying D_n with its image $\pi_n(D_\infty) \ \subset D_\infty$, that is, the image of the map $\theta_{n,\infty}$.

General Exercises on D_∞.

1. Show, for $n \leq k$ and all $a \in D_\infty$, that $\pi_n a \sqsubseteq \pi_k a \sqsubseteq a$.

2. Prove, for all $a \in D_\infty$, that $a \ = \ \bigsqcup_n \pi_n a$.

3. Using the definition of ϕ_k for $k > 0$, show inductively on k that, for all $c \in D_0$,

$$\theta_{k,\infty} \circ \theta_{0,k+1}(c) \circ \theta_{\infty,k} \ = \ \theta_{0,\infty} \circ \phi_0(c) \circ \theta_{\infty,0} \ : \ D_\infty \to D_\infty \ .$$

4. Deduce from **3** and the explicit definition of ϕ_∞ as the *lub* of a non-decreasing sequence that, for all a, b in D_∞ ,

$$(\pi_0 a) \cdot b \ = \ \theta_{0,\infty}(\phi_0(\theta_{\infty,0}(a))(\theta_{\infty,0}(b))) \ .$$

5. Deduce from **4** and Scott's version of ϕ_0 that $(\pi_0 a) \cdot b \ = \ \pi_0 a$.

6. Deduce from **5** and from $\pi_0 \perp \ = \ \perp$ that $\perp \cdot \perp \ = \ \perp$.

7. As in **3**, show inductively on $k \geq n$ that, for all $a \in D_\infty$,

$$\theta_{k,\infty} \circ \theta_{\infty,k+1}(\pi_{n+1} a) \circ \theta_{\infty,k} \ = \ \theta_{n,\infty} \circ \theta_{\infty,n+1}(a) \circ \theta_{\infty,n} \ : \ D_\infty \to D_\infty \ .$$

8. Deduce from **7** and the explicit definition of ϕ_∞ as the *lub* of a non-decreasing sequence that, for all a, b in D_∞ ,

$$(\pi_{n+1} a) \cdot b \ = \ \theta_{n,\infty}(\theta_{\infty,n+1}(a)(\theta_{\infty,n}(b))) \ .$$

(And so $(\pi_{n+1} a) \cdot b \ \in \ \pi_n D_\infty \ = \ "D_n"$.)

9. Combining **8** with $\theta_{\infty,n}(\pi_n b) = \theta_{\infty,n}(b)$, deduce that

$$(\pi_{n+1}a)\cdot b \;=\; (\pi_{n+1}a)\cdot(\pi_n b)\ .$$

10. Using **9** and **1** , deduce that, for all $k \geq n$,

$$(\pi_{n+1}a)\cdot b \;=\; (\pi_{n+1}a)\cdot(\pi_k b)\ .$$

11. Using the definition of ϕ_∞, and $a \cdot \perp = \phi_\infty(a)(\perp)$, show that, for all $a \in D_\infty$, we have

$$\pi_0 a \;\sqsubseteq\; \pi_0(a\cdot\perp)\ .$$

(Actually, equality holds here, but is unneeded below.)

Now let's come back to the situation of Park's theorem. Again to be perhaps overly fussy, we shall use x as the name for a 'general' element of D_∞. And also (to keep Λ and D_∞ distinct) underline all the elements in Λ, including variables. We fix a 'general' variable $\underline{x}$, and always use a (so-called environment) ρ which maps $\underline{x}$ to x. Fix another variable $\underline{y} \in \Lambda$, and define Curry's combinator explicitly as

$$\underline{Y} \;:=\; \lambda\underline{x}\bullet(\lambda\underline{y}\bullet\underline{x}(\underline{y}\ \underline{y}))(\lambda\underline{y}\bullet\underline{x}(\underline{y}\ \underline{y})) \;=\; \lambda\underline{x}\bullet\underline{X}\ \underline{X}\ \ ,$$

where

$$\underline{X} \;:=\; \lambda\underline{y}\bullet\underline{x}(\underline{y}\ \underline{y})\ .$$

Define elements of D_∞ by

$$Y \;:=\; \rho_+(\underline{Y}) \qquad [\text{independent of } \rho]\ ,$$

$$X \;:=\; \rho_+(\underline{X}) \qquad [\text{depending only on } \rho(\underline{x}) = x]\ .$$

Remaining Exercises to prove Park's theorem.

12. Show $\underline{X}\ \underline{z} \;=\; \underline{x}(\underline{z}\ \underline{z})$ for any variable $\underline{z}$ in Λ .

13. Deduce that $X\cdot z \;=\; x\cdot(z\cdot z)$ for any $z \in D_\infty$.

14. Show $\underline{Y}\ \underline{x} \;=\; \underline{X}\ \underline{X}$ in Λ .

15. Deduce that $Y\cdot x \;=\; X\cdot X$
[for any $x \in D_\infty$, but note that X 'depends on x'].

16. Deduce from $\pi_0 a \sqsubseteq a$ for all $a \in D_\infty$, combining **13, 6** and **11**, that

$$\pi_0 X \ \sqsubseteq \ x\cdot \bot \quad .$$

17. Deduce from **13, 5** and **16** that

$$X \cdot \pi_0 X \ = \ x \cdot (\pi_0 X \cdot \pi_0 X) \ \sqsubseteq \ x \cdot (x\cdot \bot) \quad .$$

18. Using **17** as the initial case, and noting from **10** that

$$\pi_{n+1} X \cdot \pi_{n+1} X \ = \ \pi_{n+1} X \cdot \pi_n X \ ,$$

show by induction on n that

$$X \cdot \pi_n X \ = \ x \cdot (\pi_n X \cdot \pi_n X) \ \sqsubseteq \ \phi_\infty(x)^{n+2}(\bot) \quad .$$

19. Using the definitions of "$\cdot$" and of fix, deduce from **2, 15** and **18** that

$$\phi_\infty(Y)(x) \ \sqsubseteq \ \mathsf{fix}(\phi_\infty(x)) \ .$$

20. Recalling that $\underline{Y}$ is a fixpoint combinator, show that $\phi_\infty(Y)(x)$ is a fixed point of $\phi_\infty(x)$.

21. Deduce from **20** and the basic property of fix (in **VII-8.12**) that

$$\mathsf{fix}(\phi_\infty(x)) \ \sqsubseteq \ \phi_\infty(Y)(x) \ .$$

22. Combine **19** and **21** to get Park's theorem :

$$\phi_\infty(Y) \ = \ \mathsf{fix} \circ \phi_\infty \quad .$$

Quick proofs of the crucial later exercises.

16. $\pi_0 X \ \sqsubseteq \ \pi_0(X\cdot \bot) \ \sqsubseteq \ X\cdot \bot \ = \ x \cdot (\bot \cdot \bot) \ = \ x\cdot \bot \quad .$

17. $X \cdot \pi_0 X \ = \ x \cdot (\pi_0 X \cdot \pi_0 X) \ = \ x \cdot \pi_0 X \ \sqsubseteq \ x \cdot (x\cdot \bot) \quad .$

18. The equality is **13**, and, by the hint, the left-hand side for the inductive step in the inequality is

$$x \cdot (\pi_{n+1} X \cdot \pi_n X) \ \sqsubseteq \ x \cdot (X \cdot \pi_n X) \ \sqsubseteq \ x \cdot (\phi_\infty(x)^{n+2}(\bot)) \ = \ \phi_\infty(x)^{n+3}(\bot) \quad .$$

19.

$$\phi_\infty(Y)(x) \;=\; Y\cdot x \;=\; X\cdot X \;=\; X\cdot\bigsqcup_n \pi_n X$$

$$=\; \bigsqcup_n X\cdot\pi_n X \;\sqsubseteq\; \bigsqcup_n \phi_\infty(x)^{n+2}(\bot) \;=\; \bigsqcup_n \phi_\infty(x)^n(\bot) \;=\; \mathsf{fix}(\phi_\infty(x))\ .$$

20.

$$\phi_\infty(x)(\phi_\infty(Y)(x)) \;=\; x\cdot(Y\cdot x) \;=\; \rho_+(\underline{x}(\underline{Y}\ \underline{x})) \;=\; \rho_+(\underline{Y}\ \underline{x}) \;=\; Y\cdot x \;=\; \phi_\infty(Y)(x)\,.$$

11.

$$\pi_0(a\cdot\bot) \;=\; \theta_{0\infty}(\theta_{\infty 0}(\phi_\infty(a)(\bot))) \;=\; \theta_{0\infty}\theta_{\infty 0}\bigsqcup_k(\theta_{k\infty}\circ\theta_{\infty,k+1}(a)\circ\theta_{\infty k})(\bot_\infty)$$

$$=\; \bigsqcup_k \theta_{0\infty}(\theta_{k0}(\theta_{\infty,k+1}(a)(\bot_k))) \;\sqsupseteq\; \theta_{0\infty}(\theta_{00}(\theta_{\infty,1}(a)(\bot_0)))$$

$$=\; \theta_{0\infty}(\psi_0(\theta_{\infty,1}(a))) \;=\; \theta_{0\infty}(\theta_{\infty,0}(a)) \;=\; \pi_0 a\ .$$

The step after the inequality uses the definition of ψ_0 . To strengthen the inequality to equality (which is needed further down), one shows that all terms in the *lub* just before the " $\sqsupseteq$ " agree with $\pi_0 a$, by a slightly longer argument. For example, with $k=1$, we get down to

$$\theta_{0\infty}(\theta_{10}(\theta_{\infty,2}(a)(\bot_1))) \;=\; \theta_{0\infty}(\psi_0(\theta_{\infty,2}(a)(\phi_0(\bot_0)))) \;=\; \theta_{0\infty}((\psi_0\circ\theta_{\infty,2}(a)\circ\phi_0)(\bot_0))$$

$$=\; \theta_{0\infty}(\psi_1(\theta_{\infty,2}(a))(\bot_0)) \;=\; \theta_{0\infty}(\psi_0(\psi_1(\theta_{\infty,2}(a)))) \;=\; \theta_{0\infty}(\theta_{\infty,0}(a)) \;=\; \pi_0 a\,.$$

Our early example of a Λ-term with no normal form, namely

$$(\lambda x\bullet xx)(\lambda x\bullet xx)\ ,$$

is another interesting case for the denotational semantics of Λ using Scott's D_∞ . The answer is actually that

$$\rho_+((\lambda x\bullet xx)(\lambda x\bullet xx)) \;=\; \bot\ ,$$

a theorem of Scott, derived below using Park's theorem. (This tries to say that the term carries 'no information at all'.) The reader is challenged to

use the definitions directly to show this, and see how much is involved. One approach is to attempt to show, for all $d \in D_\infty$, that

$$\rho_+((\lambda x \bullet xx)(\lambda x \bullet xx)) \ \sqsubseteq \ d \ .$$

The trick we use is the following : Let

$$I := \rho_+(\underline{I}) := \rho_+(\lambda x \bullet x) \ .$$

Using the evident fact that $\underline{I} \ \underline{A} \approx \underline{A}$ for all $\underline{A} \in \Lambda$, it is clear that $I \cdot d = d$ for all $d \in D_\infty$; that is, every element of D_∞ is a fixed point of $\phi_\infty(I)$, which is the identity map of D_∞ . In particular, the minimum fixed point of $\phi_\infty(I)$ is certainly $\perp$. And so, from Park, we see that

$$\rho_+(\underline{Y} \ I) \ = \ \perp \ .$$

Thus it suffices to prove that

$$\underline{Y} \ I \ \approx \ (\lambda x \bullet xx)(\lambda x \bullet xx) \ .$$

Letting $C := \lambda y \bullet x(yy)$, we have $\underline{Y} = \lambda x \bullet CC$, and, for any closed A, we immediately (β)-reduce $\underline{Y}A$ to $C_A C_A$, where $C_A := C^{[x \mapsto A]} = \lambda y \bullet A(yy)$. But

$$C_I \ = \ \lambda y \bullet I(yy) \approx \ \lambda y \bullet yy \ ,$$

so

$$\underline{Y} \ I \ \approx \ C_I C_I \ \approx \ (\lambda y \bullet yy)(\lambda y \bullet yy) \ ,$$

as required.

Exercise. Show that $\rho_+((\lambda x \bullet xxx)(\lambda x \bullet xxx)) \ = \ \perp$.

As mentioned earlier, by $(*)$ from about 7 pages back , we can only expect ρ_+ to be possibly injective after passing to equivalence classes under $\approx$. And it certainly won't be on non-closed terms in general unless ρ itself is injective. But for closed terms without normal forms, it still isn't injective in general, as we see further down. However, we do have the following 'faithfulness of the semantics' for terms with normal forms.

Theorem VII-9.2. *If E and F are distinct normal form terms in Λ* (i.e. they are not related by a change of bound variables)*, and continuing with Scott's D_∞ as our domain, for some ρ, we have $\rho_+(E) \neq \rho_+(F)$. In particular, the restriction of ρ_+ to closed normal form terms* (which restriction is independent of ρ) *is injective.*

This will be almost immediate from another quite deep syntactical result whose proof will be omitted:

Böhm's Theorem VII-9.3. (See [**Cu**], p.156) *If E and F are as in the previous theorem, and $y_1, \cdots, y_t$ are the free variables in EF, then there are terms $G_1, \cdots, G_t$, and $H_1, \cdots, H_k$ for some k, such that for some distinct variables u and v*

$$E^{[\vec{y}\to\vec{G}]}H_1 \cdots H_k uv \;\approx\; u \quad \text{and} \quad F^{[\vec{y}\to\vec{G}]}H_1 \cdots H_k uv \;\approx\; v \;.$$

To deduce **VII-9.2**, proceed by contradiction, and use the property labelled $(**)$ at the beginning of this section to see easily that

$$\forall\rho \;,\; \rho_+(E) = \rho_+(F) \quad \Longrightarrow \quad \forall\rho \;,\; \rho_+(E^{[\vec{y}\to\vec{G}]}) = \rho_+(F^{[\vec{y}\to\vec{G}]}) \;.$$

But then the left-hand sides in Böhm's theorem map to the same thing under all the maps ρ_+ , contradicting the fact that $\rho_+(u) \neq \rho_+(v)$ for some ρ (since ρ can take any values we want on variables).

To illustrate the 'serious' non-injectiveness of the semantics related to terms without normal forms, here is a striking theorem of Wadsworth [**Wa**].

Theorem VII-9.4. *Let $\underline{M}$ be any closed Λ-term with a normal form. Then there is a closed term $\underline{N}$ with no normal form* (so, of course, $\underline{M} \not\approx \underline{N}$) *such that $\rho_+(\underline{M}) \;=\; \rho_+(\underline{N})$.*

Proof. Firstly, let us establish that it suffices to prove this for the single example $\underline{M} = \underline{I} := \lambda x \bullet x$, where $\rho_+(\underline{I}) = I$ corresponds to the identity map $D_\infty \to D_\infty$ under ϕ_∞ . So assume (as we shall prove below) that $\rho_+(\underline{J}) = \rho_+(\underline{I})$ for some closed term $\underline{J}$ with no normal form. Given $\underline{M}$ as in the theorem, let $\underline{M}'$ be its normal form. Find the variable occurrence furthest to the right in the string $\underline{M}'$ (and say x is that variable). Now replace that single occurrence x by $(\underline{I}\; x)$ and call the result $\underline{M}''$. Similarly replace it by

($\underline{J}\ x$) and call the result $\underline{N}$. Since $\underline{M'}$ is normal, the sequence of leftmost reductions for $\underline{N}$ is simply the same (infinite) sequence for $\underline{J}$ applied to that subterm. So $\underline{N}$ has no normal form. Also

$$\rho_+(\underline{M}) \ = \ \rho_+(\underline{M'}) \ = \ \rho_+(\underline{M''}) \ ,$$

since $\underline{M} \ \approx \ \underline{M'} \ \approx \ \underline{M''}$, the latter because $\underline{I}\ x \approx x$. But also

$$\rho_+(\underline{M''}) \ = \ \rho_+(\underline{N}) \ ,$$

completing this part of the proof. To see this last equality, note that, in the notation way back in **VII-1.1**, it is easy to see that

$$\text{if} \ \ \rho_+(A) \ = \ \rho_+(B) \ , \ \ \text{then} \ \ \rho_+(SAT) \ = \ \rho_+(SBT) \ .$$

So take A and B to be $\underline{I}$ and $\underline{J}$, respectively, in $SAT = \underline{M''}$ and $SBT = \underline{N}$.

Thus it remains to establish the particular case $\underline{M} = \underline{I}$. Here we'll give $\underline{J} = \underline{N}$ quite explicitly. Let

$$\underline{F} \ := \ \lambda fxy \bullet x(fy) \ , \quad \text{and take} \quad \underline{J} \ := \ \underline{Y}\ \underline{F} \ ,$$

where $\underline{Y}$ is again Curry's fixpoint combinator. We shall show quite generally in the paragraphs after next that, for any closed $\underline{A} \in \Lambda$,

$$\underline{F}\ \underline{A} \ \approx \ \underline{A} \implies \ \rho_+(\underline{A}) \ = \ \rho_+(\underline{I}) \qquad\qquad (*)$$

(It is very easy, but irrelevant, to see that $\underline{F}\ \underline{I} \ \approx \ \underline{I}$.) However, the fact that $\underline{F}\ \underline{J} \ \approx \ \underline{J}$ is immediate from the fixpoint property of Curry's combinator, so the above will give us $\rho_+(\underline{J}) \ = \ \rho_+(\underline{I})$, the main conclusion of the theorem.

First let's prove the other required property of $\underline{J}$, namely that it has no normal form. With $\underline{Y} := \lambda x \bullet C_x C_x$, where $C_x := \lambda y \bullet x(yy)$ and

$$C_A \ := \ C_x^{[x \mapsto A]} \ = \ \lambda y \bullet A(yy) \ \ ,$$

the leftmost reduction of $\underline{J} = \underline{Y}\ \underline{F}$ goes as follows :

$$\underline{Y}\ \underline{F}\ \overline{\succeq}\ C_{\underline{F}}C_{\underline{F}}\ =\ (\lambda y \bullet \underline{F}(yy))C_{\underline{F}}\ \overline{\succeq}\ \underline{F}(C_{\underline{F}}C_{\underline{F}})$$
$$\overline{\succeq}\ \lambda xy \bullet x((C_{\underline{F}}C_{\underline{F}})y)\ \overline{\succeq}\ \cdots\cdots$$
$$\overline{\succeq}\ \lambda xy \bullet x((\lambda xy \bullet x((C_{\underline{F}}C_{\underline{F}})y))y)\ \overline{\succeq}\ \cdots\cdots$$
$$\overline{\succeq}\ \lambda xy \bullet x((\lambda xy \bullet x((\lambda xy \bullet x((C_{\underline{F}}C_{\underline{F}})y))y))y)\ \overline{\succeq}\ ------\ .$$

This sequence does not terminate, as required.

The subtlest part comes now, in proving $(*)$ above, by manipulations due to Wadsworth which seem fiendishly clever to me! So assume that $\underline{A}$ is any closed term with $\underline{F}\ \underline{A}\ \approx\ \underline{A}$, and we shall show that $A = I$, where $A := \rho_+(\underline{A})$. For all $\underline{B}$ and $\underline{C}$ in Λ, we have

$$\underline{A}\ \underline{B}\ \underline{C}\ \approx\ \underline{F}\ \underline{A}\ \underline{B}\ \underline{C}\ =\ (\lambda fxy \bullet x(fy))\underline{A}\ \underline{B}\ \underline{C}\ \approx\ \underline{B}\ (\underline{A}\ \underline{C})$$

Applying ρ_+ , we get, for all B and C in D_∞ ,

$$A \cdot B \cdot C\ =\ B \cdot (A \cdot C) \qquad\qquad (**)$$

To show that $A = I$, we'll now use only $(**)$. It suffices to show that $\pi_n A = \pi_n I$ for all n, by induction on n, involving the projections π_n introduced back in the discussion of Park's theorem. Here is a repeat of three of the exercises from back there, plus a list of four new general exercises for the reader to work on. But if needed, see after the end below of the present theorem's proof for some hints which make their proofs completely mechanical.

5. $(\pi_0 a) \cdot b\ =\ \pi_0 a$.

9. $(\pi_{n+1} a) \cdot b\ =\ (\pi_{n+1} a) \cdot (\pi_n b)$.

11. $\pi_0 a\ =\ \pi_0(a \cdot \bot)$.

23. $\pi_n \bot\ =\ \bot$.

24. $\bot \cdot x =\ \bot$.

25. $I \cdot C =\ C$.

26. $(\pi_{n+1} a) \cdot b\ =\ \pi_n(a \cdot \pi_n b)$.

In both initial cases and also in the inductive case, we'll use the extensionality of D_∞ twice.

The initial case $n = 0$:

Using **5**, then **5**, then **11**, for all B, x and y in D_∞ ,

$$\pi_0 B \cdot x \cdot y \;=\; \pi_0 B \cdot y \;=\; \pi_0 B \;=\; \pi_0(B \cdot \bot) \;.$$

Thus, with $B = I$, using the above, then **25**,

$$\pi_0 I \cdot x \cdot y \;=\; \pi_0(I \cdot \bot) \;=\; \pi_0 \bot \;\;.$$

With $B = A$, using the above, then **11**, then $(**)$, then **24**,

$$\pi_0 A \cdot x \cdot y \;=\; \pi_0(A \cdot \bot) \;=\; \pi_0(A \cdot \bot \cdot \bot) \;=\; \pi_0(\bot \cdot (A \cdot \bot)) \;=\; \pi_0 \bot \;\;.$$

So $\pi_0 A = \pi_0 I$, by extensionality.

The initial case $n = 1$:

Using **26**, then **25**, then idempotency of π_0, then **5**,

$$\pi_1 I \cdot x \cdot y \;=\; \pi_0(I \cdot \pi_0 x) \cdot y \;=\; \pi_0(\pi_0 x) \cdot y \;=\; \pi_0 x \cdot y \;=\; \pi_0 x \;.$$

Using **26**, then **5**, then **11**, then $(**)$, then **5**, then idempotency of π_0,

$$\begin{aligned} \pi_1 A \cdot x \cdot y \;&=\; \pi_0(A \cdot \pi_0 x) \cdot y \;=\; \pi_0(A \cdot \pi_0 x) \\ &=\; \pi_0(A \cdot \pi_0 x \cdot \bot) \;=\; \pi_0(\pi_0 x \cdot (A \cdot \bot)) \;=\; \pi_0(\pi_0 x) \;=\; \pi_0 x \;. \end{aligned}$$

So $\pi_1 A = \pi_1 I$, by extensionality.

The inductive case :

Assume inductively that $\pi_{n+1} A = \pi_{n+1} I$. Using **26**, then **25**, then idempotency of π_{n+1}, then **9**,

$$\pi_{n+2} I \cdot x \cdot y \;=\; \pi_{n+1}(I \cdot \pi_{n+1} x) \cdot y \;=\; \pi_{n+1}(\pi_{n+1} x) \cdot y \;=\; \pi_{n+1} x \cdot y \;=\; \pi_{n+1} x \cdot \pi_n y \;.$$

Now using **26**, then the inductive assumption, then **26**, then **25**, then idempotency of π_n,

$$\pi_n(A \cdot \pi_n y) \;=\; \pi_{n+1} A \cdot y \;=\; \pi_{n+1} I \cdot y \;=\; \pi_n(I \cdot \pi_n y) \;=\; \pi_n(\pi_n y) \;=\; \pi_n y \;.$$

So, using **26**, then **26**, then $(**)$, then **9**, then **26**, then idempotency of π_{n+1}, then the display just above,

$$\pi_{n+2}A \cdot x \cdot y \;=\; \pi_{n+1}(A \cdot \pi_{n+1}x) \cdot y \;=\; \pi_n(A \cdot \pi_{n+1}x \cdot \pi_n y)$$

$$=\; \pi_n(\pi_{n+1}x \cdot (A \cdot \pi_n y)) \;=\; \pi_n(\pi_{n+1}x \cdot \pi_n(A \cdot \pi_n y))$$

$$=\; \pi_{n+1}(\pi_{n+1}x) \cdot \pi_n(A \cdot \pi_n y) \;=\; \pi_{n+1}x \cdot \pi_n(A \cdot \pi_n y) \;=\; \pi_{n+1}x \cdot \pi_n y \;.$$

So, once again by extensionality, $\pi_{n+2}A = \pi_{n+2}I$.

Thus **VII-9.4** is now finally proved.

As for the new exercises used above, if you haven't done them already, here are statements which reduce them to mechanical checks :

As for **23**, this just amounts to the fact that

$$\bot \;:=\; \bot_{D_\infty} \;=\; (\bot_0, \bot_1, \bot_2, \cdots) \;,$$

since the right-hand side is clearly the smallest element in D_∞ .

As for **24**, the argument is the same as for **6**.

As for **25**, this is immediate from $\underline{I}\;\underline{C} \;\approx\; \underline{C}$.

27. Using the definition of the maps ψ_j, show by induction on $k \geq n$ that for all $a \in D_\infty$ and $y \in D_n$, we have

$$\theta_{kn}(\theta_{\infty,k+1}(a)(\theta_{nk}(y))) \;=\; \theta_{\infty,n+1}(a)(y) \;.$$

As for **26**, use **27** with $y = \theta_{\infty n}(b)$ in the last step just below to see that

$$\pi_n(a \cdot \pi_n b) \;=\; \pi_n(\phi_\infty(a)(\pi_n b)) \;=\; \theta_{n\infty} \circ \theta_{\infty n}(\bigsqcup_k \theta_{k\infty}(\theta_{\infty,k+1}(a)(\theta_{\infty,k} \circ \theta_{n\infty} \circ \theta_{\infty n}(b))))$$

$$=\; \theta_{n\infty}(\bigsqcup_k \theta_{kn}(\theta_{\infty,k+1}(a)(\theta_{nk} \circ \theta_{\infty n}(b)))) \;=\; \theta_{n\infty}(\theta_{\infty,n+1}(a)(\theta_{\infty n}(b))) \;.$$

But the latter is $(\pi_{n+1}a) \cdot b$ by **8** .

Quite a bit easier is producing an example of two distinct elements of $\Lambda/\approx$, both closed, and *neither* with normal form, which give the same

element of D_∞ under ρ_+ . Such elements are the classes of $\underline{Y}$ and $\underline{Y}' = \underline{Y}\ \underline{G}$, where $\underline{Y}$ is again Curry's fixpoint combinator, and

$$\underline{G} \ := \ \lambda uv \bullet v(uv) \ .$$

The argument is quite elegant :

Firstly, it's becoming slightly embarrassing how many syntactic results we are leaving the reader to look up, but here is the final one:

Proposition VII-9.5. $\underline{Y}' \not\approx \underline{Y}$ *and neither has a normal form.*
See Böhm p. 179 in **[St-ed]**.

To show that $\rho_+\underline{Y}' = \rho_+\underline{Y}$, first we establish a nice result of independent interest.

Proposition VII-9.6.

$$\forall \underline{Z} \in \Lambda \ , \ [\ \underline{G}\ \underline{Z} \ \approx \ \underline{Z} \ \iff \ \forall \underline{A} \in \Lambda \ , \ \ \underline{A}\ (\underline{Z}\ \underline{A}) \ \approx \ \underline{Z}\ \underline{A}\ ] \ .$$

That is, $\underline{Z}$ *is a fixed point of* $\underline{G}$ *if and only if it is a fixpoint operator.*

Proof. First note that

$$\underline{G}\ \underline{Z} \ = \ (\lambda uv \bullet v(uv))\underline{Z} \ \approx \ \lambda v \bullet v(\underline{Z}v) \qquad\qquad (*)$$

As for $\Longrightarrow$: Using the assumption, then $(*)$, then β-reduction,

$$\underline{Z}\ \underline{A} \ \approx \ \underline{G}\ \underline{Z}\ \underline{A} \ \approx \ (\lambda v \bullet v(\underline{Z}v))\underline{A} \ \approx \ \underline{A}\ (\underline{Z}\ \underline{A}) \ \ .$$

As for $\Longleftarrow$: Using $(*)$, then the assumption with $\underline{A} = v$, then η-reduction,

$$\underline{G}\ \underline{Z} \ \approx \ \lambda v \bullet v(\underline{Z}v) \ \approx \ \lambda v \bullet \underline{Z}v \ \approx \ \underline{Z} \ \ .$$

Now abbreviate $\rho_+\underline{Y}'$ and $\rho_+\underline{Y}$ to Y' and Y, respectively.

Proposition VII-9.7. $\underline{Y}'$ *is a fixpoint operator ; and so*

$$A \cdot (Y' \cdot A) = Y' \cdot A \ \text{ for all } \ A \in D_\infty \ .$$

Proof. Use 9.6$\Longrightarrow$, after calculating using the fact that $\underline{Y}$ is a fixpoint operator :

$$\underline{G}\ \underline{Y}' \ = \ \underline{G}(\underline{Y}\ \underline{G}) \ \approx \ \underline{Y}\ \underline{G} \ = \ \underline{Y}' \ .$$

Corollary to Park's Theorem VII-9.8.

$$Y \cdot A \sqsubseteq Y' \cdot A \quad \text{for all } A \in D_\infty .$$

Proof. Both $Y \cdot A$ and $Y' \cdot A$ are fixed by $A \cdot$. But Park says that $Y \cdot A$ is the smallest of all elements in D_∞ fixed by $A \cdot$.

Corollary VII-9.9 (of $\Longleftarrow$ in 9.6).

$$\underline{G}\,\underline{Y} \approx \underline{Y} \quad \text{and so} \quad G \cdot Y = Y$$

Thus Y is a fixed point of $G \cdot$. And so $Y \cdot G \sqsubseteq Y$, i.e. $Y' \sqsubseteq Y$, since $Y \cdot G$ is the smallest fixed point of $G \cdot$. Thus we get

Corollary VII-9.10

$$Y' \cdot A \sqsubseteq Y \cdot A \quad \text{for all } A \in D_\infty .$$

And now finally

Corollary VII-9.11 (of 9.10 and 9.8)

$$Y' \cdot A = Y \cdot A \quad \text{for all } A \in D_\infty .$$

Thus, by extensionality, we have what we want, namely

$$Y' = Y \quad \text{that is,} \quad \rho_+\underline{Y'} = \rho_+\underline{Y} .$$

We should finish with what seems to be the canonical example used whenever anything related to 'recursive computing' needs a simple illustration, namely, the factorial function. And then again, maybe we shouldn't, though just seeing how complicated or otherwise a Λ-term is needed to programme the factorial function is a bit interesting!

VIII—Floyd-Hoare Logic

We discuss here a rather syntactic approach to program *correctness* (or *program verification*), beginning with the austere 'language' **ATEN**, and then discussing more realistic programming languages. Later this will all be compared and contrasted with the approach of denotational semantics. In fact, a careful verification of the soundness of the proof system in the syntactic approach would often use a denotational specification of the semantics, although we shall use a more operational semantics in each case; for **ATEN**, just the one originally given. There is also a point of view which sees the F-H approach as an alternative form of specifying the semantics of a programming language.

8.1—Floyd-Hoare Logic for ATEN.

Floyd-Hoare logics are deductive systems for deriving statements of the form $F\{C\}G$, where F and G are $1^{\underline{\text{st}}}$order formulas (the 'pre- and post-conditions'), and C is a command from some imperative language. Here we'll choose that language to be **ATEN**. This choice will probably seem far too simplistic to any reader with experience of serious work in F-H logic, which exists mainly for more practical (and therefore complicated) languages, ones which include features such as procedure declarations and calls. See [**C**]. We shall study such a language in Subsection 8.3. In principle, of course, **ATEN** computes anything that is computable. In any case, commenting about soundness and adequacy (completeness) in the case of a simple language like **ATEN** serves to emphasize all the main general points, without having intricacies which obscure them.

But what do we 'mean' by a string $F\{C\}G$ of symbols? The idea is that this string makes an assertion. That assertion is this : *the formula G will be true for the state resulting from executing the command C, as long as we start from a state for which F is true and for which the execution of C does indeed terminate.* So it will come as no surprise that here, the $1^{\underline{\text{st}}}$order language from which F and G come is $1^{\underline{\text{st}}}$order number theory. In the next subsection, we shall discuss a more general situation with **ATEN**, as well as languages with more complicated features.

While **ATEN** is the command language used, it will also be no surprise that, in the semantics, we shall concentrate a great deal on **N** as the interpretation.

But to begin, it is better to consider an arbitrary interpretation I of the $1^{\underline{\text{st}}}$order number theory language. So I is a non-empty set, together with interpretations of the symbols $+$, $\times$, $<$, 0 and 1 over that set. We won't worry about how the elements of I (real numbers, for example) would be represented in some machine which we think of as implementing **ATEN** commands. Nor will we be concerned about implementing the operations which interpret the symbols $+$ and $\times$.

Definition. The F-H statement $F\{C\}G$ is *true in interpretation* I if and only if, for all $\underline{v}$ such that F is true at $\underline{v}$ and such that, when executed with initial state $\underline{v}$, the command C terminates (say, with state $\underline{w}$), we have that G is true at $\underline{w}$.

So this definition merely makes more formal the earlier explanation of the Floyd-Hoare statement $F\{C\}G$. Recall from [**LM**] that $\underline{v} = (v_0, v_1, v_2, \cdots)$, and also $\underline{w}$, are infinite sequences from the interpretation I. We 'store v_i in bin $\#i$ ', for the execution of C. Note also that F and G are full-blown formulas, not necessarily quantifier-free as with the formulas that are used 'for control' within whdo-commands.

The formulas F and G are also likely to have free variables, since if neither did, the definition above would say little about the behaviour of the command C. More precisely, it would say that, if C is a command with at least one input state where its execution terminates, then the sentence F being true in I implies that the sentence G is true in I (and it would put no restrictions at all on F and G if C 'loops' on all inputs). (There is a text which exposits $1^{\underline{\text{st}}}$order logic, religiously disallowing free variables in formulas, with considerable resulting awkwardness. It seems ironic, in view of the above, that the author is a computer scientist, and the intended audience consists of CSers.)

Note once again that termination of the program is part of the assumptions, not the conclusion in the definition. This is called *partial correctness* to contrast it with *total correctness*, where termination is part of the conclusion. Because of the results of Turing about the halting problem (see [**CM**]), thinking about systems for total correctness brings forward some serious limitations right at the beginning. There is a considerable literature on that topic, but we will only touch on it in the second addendum below. Expressed symbolically in terms from the beginning of [**CM**], a total correctness statement would be saying

$$F \text{ true at } s \implies ||C||(s) \neq err \ \wedge \ G \text{ true at } ||C||(s) \ .$$

The definition above for partial correctness says that truth of $F\{C\}G$ amounts to, for all s,

$$F \text{ true at } s \ \wedge \ ||C||(s) \neq err \quad \Longrightarrow \quad G \text{ true at } ||C||(s) \ .$$

Before getting into the proof system, possibly the reader is puzzled about one thing—surely the standard naive thought about correctness of programs is that the state after execution should be properly related to the state *before* execution? After all, we have emphasized, as most do at an elementary level, the idea that a command (or program) is a recipe for computing a function from states to states. But an F-H statement seems to be only asserting something about 'internal' relations involving the output state, not any relation of it to the input state. The way to deal with this question is to refer to snapshot variables (or history, or ghost, or auxiliary variables)—an example is the easiest way to explain this. Recall the **ATEN** command C whose purpose was to interchange x_1 and x_2. We used bin 3 as an intermediate storage, but the command didn't use any x_i for $i > 3$, so we'll employ x_4 and x_5 as our snapshot variables. An F-H statement whose truth should convince most people of the correctness of C would then be the following :

$$(x_4 \approx x_1 \ \wedge \ x_5 \approx x_2) \ \{C\} \ (x_4 \approx x_2 \ \wedge \ x_5 \approx x_1) \ .$$

Admittedly, this says nothing, strictly speaking, about what happens when we execute C without first making sure that bins 4 and 5 have identical contents to bins 1 and 2 respectively. But it seems that one simply has to *accept* as a kind of meta-semantic fact the obvious statement that the content of bin i is unchanged and irrelevant to the computation, if x_i does not occur in the command. Only believers in voodoo would waste time questioning that.

You can find a number of extra examples of specific F-H derivations in the first section of [**G**]. An exercise would be to deduce the completeness of the system there from that of the one below.

Definition. The deductive system we shall use is given symbolically as follows :

$$\text{(I)} \qquad \frac{F\{C\}G \ , \ G\{D\}H}{F\{(C;D)\}H}$$

(II) $$\frac{\text{empty}}{F^{[x\to t]}\{x \doteq t\}F}$$

(III) $$\frac{(F \wedge H)\{C\}F}{F\{\mathsf{whdo}(H)(C)\}(F \wedge \neg H)}$$

(IV) $$\frac{F \to F' \ , \ F'\{C\}G' \ , \ G' \to G}{F\{C\}G}$$

Comments. Each of these is to be regarded as a *rule of inference*, for use in *derivations*. The latter are finite sequences of F-H statements and of formulas $(A \to B)$, ending with an F-H statement. But the rule conclusions $(\frac{}{\text{under the line}})$ are all F-H statements, so we'll need some discussion (below) about how lines $(A \to B)$ can appear in such a derivation. In any case, only rule (IV) uses such formulas. Rule (III) contains a formula H which must be quantifier-free, since it occurs within a whdo-command. And rule (II) may be better regarded as a 'logical' axiom, since it has no premiss.

Definition. A rule $\frac{\alpha_1,\alpha_2,\cdots,\alpha_k}{\beta}$ is *valid* in I if and only if β is true in I whenever all α_i are true in I (as per the above definition for truth of F-H statements, or see [**LM**], p.212-214, for those α_i which are $1^{\underline{\text{st}}}$order formulas).

Definition. A system is *sound* for I if and only if all its rules are valid in I. (But the defined system is written with no dependence on any particular interpretation I, so it is simply *sound* (period!), as the theorem below states. In fact, it has no dependence even on which $1^{\underline{\text{st}}}$order language is used, as we discuss in the following subsection.)

Remark. There is surely no need to get formal about derivations, nor about the formulation, much less the proof, of the fact that all the F-H lines are themselves true in I, in a derivation using a sound system and only premisses that are true in I; in particular the *conclusion* of that derivation (its last line) is true in I. Most derivations have no premisses, certainly ones purporting to show that a particular program is correct. See also the second addendum below, where a discussion of including strings with propositional

connectives, and where we take the viewpoint that the unruly set of all true (in **N**) $1^{\underline{\text{st}}}$order formulas is actually the set of premisses.

Theorem 8.1. *The previous system is sound, as long as rule* (IV) *uses only formulas* $(A \to B)$ *which are true in the interpretation considered.*

Proof. Each rule is pretty obviously valid, but we'll give some details below. This is written out

(1) to give the reader some practice with the definitions; and

(2) since it is a matter of experience in this subject that unsound systems have been published several times, for the more complicated practical languages as in Subsection 8.3, and even for a system using **ATEN**, where one extends the F-H statements by using propositional connectives, as we do in the second addendum below, where these statements are called *assertions*.

See also the four quotes in the 4th subsection, just before the addenda.

Another reason to write this out is to avoid the need to do so later for analogous rules in a much more complicated system for a closer to practical, but messier, command language.

Note also that until we get more explicit about the formulas $(A \to B)$, and even then, one has a particular interpretation in mind when talking about derivations, which is often not the case in pure logic.

For rule (I), if F is true at $\underline{v}$ and $(C;D)$ is executed starting with $\underline{v}$, then we have the following possibilities.

(i) If C loops at $\underline{v}$, then so does $(C;D)$.

(ii) If not, let C output $\underline{w}$ when its input is $\underline{v}$. Then G is true at $\underline{w}$, since we are assuming that $F\{C\}G$ is true.

(iii) Now if D loops at $\underline{w}$, then so does $(C;D)$ at $\underline{v}$.

(iv) If not, let D output $\underline{z}$ when its input is $\underline{w}$. Then H is true at $\underline{z}$, since we are assuming that $G\{D\}H$ is true. But now, $(C;D)$ outputs $\underline{z}$ when its input is $\underline{v}$, so we have shown $F\{(C;D)\}H$ to be true, as required.

For rule (II), if $F^{[x_i \mapsto t]}$ is true at $\underline{v}$, then F is true at $(v_0, v_1, \cdots, v_{i-1}, t^{\underline{v}}, v_{i+1}, \cdots)$, that is, F is true at $\underline{w}$, where the latter is the output state when $x_i \leftarrow\!\!\!\div\ t$ is executed on $\underline{v}$, as required. See **[LM]**, pp.211, 226-7, if necessary, for $t^{\underline{v}}$.

For rule (III), assume that H is quantifier-free, that $(F \wedge H)\{C\}F$ is true, and that F is true at $\underline{v}$. Suppose also that $\mathsf{whdo}(H)(C)$ terminates with $\underline{w}$, when executed on $\underline{v}$. Then H is false at $\underline{w}$, so

(A) : $\neg H$ *is true there.*

Also H is true before each execution of C within the execution of $\mathsf{whdo}(H)(C)$.

Then, inductively, F is also true before each such execution, using the truth of $(F \wedge H)\{C\}F$ each time. Using it one last time,

(B) : *F is true after execution of* $\mathsf{whdo}(H)(C)$.

Thus, by (A) and (B), $F \wedge \neg H$ is true after execution of $\mathsf{whdo}(H)(C)$, as required.

Finally, for rule (IV), suppose that $\underline{v}$ is such that F is true there and C terminates when executed with $\underline{v}$ as input. Then F' is also true there, since we are assuming $F \to F'$ is true in I. But now, since we are assuming $F'\{C\}G'$ is true, we see that G' is true after executing C, and so, as required, G also is, since $G' \to G$ is true in I.

Now we shall begin to consider *adequacy* of the system, the reverse of soundness. Is it possible that any true F-H statement $F\{C\}G$ can be derived within the deductive system which we have given? Well, the system hasn't exactly been given, since the formulas $(A \to B)$ allowed in rule (IV) need to be specified. We certainly want them at least to be true in I, as in the soundness theorem. But a fundamental theme from post-1930 $1^{\underline{\text{st}}}$order logic, that *deducibility* can be much weaker than *truth*, has now to be considered.

For the sake of concreteness, from here on in this subsection we shall stick to $\mathbf{N}$ as the interpretation. First let us suppose that we tack on (to the given F-H proof system) some proof system for $1^{\underline{\text{st}}}$order number theory formulae which is sound with respect to the interpretation $\mathbf{N}$.

(For example, the formula $\neg 0 \approx x \ \to \ 0 < x$ might be derivable for use in rule (IV) of the F-H system, a formula which is true in $\mathbf{N}$, but certainly is not logically valid. Indeed, we might take it as one of the axioms for the system.) This $1^{\underline{\text{st}}}$order proof system is expected to be *decidable* or *axiomatic* in that there is, for example, an algorithm for recognizing whether a given formula is an axiom of the system.

Since the F-H system we gave is also decidable, it is a standard consequence that the F-H statements which can be derived using the combined system form a recursively enumerable set (with respect to some Gödel numbering of the set of such statements).

Now consider the derivable statements of the special type as follows :

$$0 \approx 0\{C\}\neg 0 \approx 0 \ .$$

Clearly one can 'automatically' recognize such formulas within an enumera-

tion as above, so we conclude that the derivable formulas of that type form a recursively enumerable set.

But what about the set of all such statements which happen to be true in **N**? It is clear, since $\neg t \approx t$ tends to be false in any interpretation of any $1^{\underline{\text{st}}}$order language (with equality), that $0 \approx 0\{C\}\neg 0 \approx 0$ is true for exactly those commands C which fail to terminate (or *halt*) no matter what the input state is.

For a contradiction, suppose now that every F-H statement which is true in **N** can be derived using our combined system. It follows that the set of C which always fail to halt must be recursively enumerable. But this directly contradicts a well-known theorem—see for example [**CM**], pp.106-7 where that set is shown not to be decidable. But the complement of that set is easily seen to be recursively enumerable (it can be semi-decided by just trying, computational step-by-step, all possible inputs for the command, until and if one is found where it terminates). So the set itself cannot be r.e., since, by [**CM**], **IV-5.3**, decidability is implied for an r.e. set whose complement is also r.e. This contradiction now shows that our combined system is never complete, whatever axiomatic proof system for number theory we use to supplement the four rules forming the F-H system.

For logicians, there is a rather more famous non-r.e. set, namely the set of formulas in $1^{\underline{\text{st}}}$order number theory which are true in **N**. (See, for example, Appendix LL in [**LM**].) One can base an alternative argument to the above on that non-r.e. set as follows. First fix a formula T which is always true, for example, $0 \approx 0$. Now fix a command, $NULL$, which does nothing other than terminate for every input. ('It computes the identity function.') An example might be $x_0 \leftarrow\!\!\div\ x_0$. Now consider the set of all F-H statements of the form $T\{NULL\}G$. Such a statement is true in **N** precisely when G is. So the set of such would be r.e. if we had a combined axiomatic proof system which was complete, and so no such system can exist.

Note that the above arguments depend only on both the F-H proof system for F-H statements and the proof system for $1^{\underline{\text{st}}}$order formulas being *axiomatic*, i.e., the rules and axioms form *decidable* sets. They have no dependence on the particular system defined above, or really even on the choice of **ATEN** as a 'Turing-equivalent' command language. Thus we have

Theorem 8.2. *For no combined axiomatic proof system, using a F-H system involving* $1^{\underline{\text{st}}}$*order formulas combined with a proof system producing formulas true in* **N**, *can we have adequacy (i.e. completeness). That is, some true F-H statements will necessarily be underivable by the combined system.*

So, to hope for completeness, we must have something like a 'non-axiomatic' system for deriving true (in **N**) formulas $(A \to B)$ as input for the final rule in our F-H system. I don't know of any such system which is fundamentally different from simply, in oracle-like fashion, assuming that all (true) such formulas can be used as required. So we'll call this the *oracular F-H system.* A bit surprising (at least to me) is the fact that we now do get completeness—somehow, rule (III), for whdo-commands, does say everything that needs to be said, though that seems to me rather unobvious.

In [**C**], this idea of a non-axiomatic system is expressed that way. However, it is probably best to just regard all derivations in the F-H system as having a set of premisses consisting of all $1^{\underline{\text{st}}}$order formulas which are true in **N**. So we imagine the oracle as being able to decide truth in **N**, not just being able to list all true formulas in **N**. Thus the set of all derivations becomes 'decidable relative to the set of all true formulas in **N**'. And so the set of all deducible F-H statements becomes 'recursively enumerable relative to the set of all true formulas in **N**'. This and the previous singly quoted phrase have precise meanings in recursion theory.

To prove completeness, we need a new concept, the *post relation*, and one good theorem about it, namely, its $1^{\underline{\text{st}}}$order definability.

Definition. Let F be a $1^{\underline{\text{st}}}$order number theory formula. and let C be a command from **ATEN**. Let $(y_1, \cdots, y_n)$ be a list of distinct variables which includes all the variables occurring in C and/or occurring freely in F. Let $Q_{F,C}^{(y_1,\cdots,y_n)}$ be the n-ary relation on natural numbers defined by

$$Q_{F,C}^{(y_1,\cdots,y_n)}(d_1, \cdots, d_n) \iff$$

for at least one input $(d'_1, \cdots, d'_n)$ for which F is true [with $y_i = d'_i$], the command C terminates with output $(d_1, \cdots, d_n)$.

Theorem 8.3. *For any F and C, the relation $Q_{F,C}^{(y_1,\cdots,y_n)}$ is '*$1^{\underline{\text{st}}}$*order definable', in that there is a formula $G = G_{F,C}$ with no free variables other than $y_1, \cdots, y_n$ such that G is true at $(d_1, \cdots, d_n)$ if and only if $Q_{F,C}^{(y_1,\cdots,y_n)}(d_1, \cdots, d_n)$.*

This follows using a major result, essentially due to Gödel, given in [**CM**]

as **V-2.3**, p.175. Take G to be

$$\exists z_1 \cdots \exists z_n \ (F \wedge H) \ ,$$

where $z_1, \cdots, z_n$ are distinct variables, disjoint from $\{y_1, \cdots, y_n\}$, and H is a formula with free variables from the y_i and z_i, such that

$$H^{[\vec{y} \to \vec{d} \ , \ \vec{z} \to \vec{d'}]} \text{ is true in } \mathbf{N} \qquad \Longleftrightarrow \qquad ||C||(\vec{d'}) = \vec{d} \, .$$

The relation on the right-hand side in the display (that is, with input $\vec{d'}$, command C terminates with output $\vec{d}$) is clearly semi-decidable, so Gödel's result assures us that H exists, i.e. the relation is $1^{\underline{\text{st}}}$order definable. It is straightforward to check that G has the required property. (Readers might give the details as an exercise—compare to the details we give for the expressivity result in the second addendum.)

In the next subsection, we'll begin to consider more general $1^{\underline{\text{st}}}$order languages, and associated command languages. We'll refer to such a setup as *expressive* when the statement in **8.3** holds. That implies, for each (F, C), the existence of a formula $G = G_{F,C}$ such that

(i) $F\{C\}G_{F,C}$ is true; and

(ii) for all formulas H, we have that $F\{C\}H$ is true implies that $G_{F,C} \to H$ is true.

More elegantly, this can be stated as

for all formulas H, we have [$F\{C\}H$ is true iff $G_{F,C} \to H$ is true].

(A formula with the properties of $G_{F,C}$ is often referred to as a "strongest postcondition". Pressburger arithmetic for $\mathbf{N}$ is an example of a non-expressive language—just drop the function symbol for multiplication from the language. A famous result says that truth in $\mathbf{N}$ then becomes decidable.)

Now we can state and prove the main result of this section.

Theorem 8.4. *The oracular F-H proof system is* complete*, in that every F-H statement which is true in* $\mathbf{N}$ *can be derived using that system.*

Remark. The statement $F\{C\}G$ is clearly true in each of the following cases. So the theorem guarantees the existence of a derivation. As an exercise, give a more direct argument for the derivation each time.

(i) G is logically valid; any F and C.

(ii) F is the negation of a logically valid formula; any G and C.

(iii) C 'loops' on every input; any F and G.

Actually, (iii) is needed later, and seems not completely obvious. The best I can come up with is this : By (IV), we need only find a derivation for $0 \approx 0\{C\}\neg 0 \approx 0$. Clearly C is not an assignment command. When C has the form $(D;E)$, go through that case of the proof below. When C is $\mathsf{whdo}(H)(D)$, it's certainly simpler than that case of the proof below. Firstly, H is necessarily true for all $\underline{v}$, as we use in the last sentence below. Now (i) gives a derivation for $0 \approx 0\{D\}H$, and hence for $(H \wedge H)\{D\}H$, by (IV), since $H \wedge H \rightarrow 0 \approx 0$ is true. So the while-rule gives one for $H\{\mathsf{whdo}(H)(D)\}(H \wedge \neg H)$. But $0 \approx 0 \rightarrow H$ and $H \wedge \neg H \rightarrow \neg 0 \approx 0$ are both true, so apply (IV) again to finish.

Proof. We proceed by structural induction on $C \in \mathbf{ATEN}$, assuming $F\{C\}G$ to be true in $\mathbf{N}$, and showing how to get a derivation of it. The initial and easier inductive cases will be done first to help orient the reader. The harder inductive case (namely when C is a whdo-command) is a substantial argument, due to Cook [**C**] (though it is far less intricate than his arguments when we have a more serious programming language with procedure calls, as Subsection 8.3 will explain).

Initial case, where $C = \ x \doteleftarrow t$: Assume that $F\{x \doteleftarrow t\}G$ is true in $\mathbf{N}$. Then "G with all free occurrences of x replaced by the term t" is true wherever F is true; that is, $F \rightarrow G^{[x \rightarrow t]}$ is true in $\mathbf{N}$. So the semantically complete (oracular) system for $\mathbf{N}$ gives a derivation of the latter formula. Now use rule (II) to get a derivation of $G^{[x \rightarrow t]}\{x \doteleftarrow t\}G$. To finish, apply rule (IV) as follows:

$$\frac{F \rightarrow G^{[x \rightarrow t]} \ , \ G^{[x \rightarrow t]}\{x \doteleftarrow t\}G \ , \ G \rightarrow G}{F\{x \doteleftarrow t\}G}$$

Inductive case, where $C = \ (D;E)$: Assume $F\{(D;E)\}H$ is true in $\mathbf{N}$. Let G express the post relation for (F,D) . By definition, $F\{D\}G$ is true in $\mathbf{N}$. We'll show $G\{E\}H$ is also true in $\mathbf{N}$, so, by the inductive assumption, they both have derivations within our oracular system, and then rule (I) gives the required result. Given a state $\vec{d}$ where G is true, its definition guarantees a state $\vec{d}'$ where: F is true and D 'produces' $\vec{d}$. But then, if E 'produces' $\vec{d}''$ with input $\vec{d}$, we must have that H is true at $\vec{d}''$, as required, since $(D;E)$ produces $\vec{d}''$ from input $\vec{d}'$, and $F\{(D;E)\}H$ is given to be true.

Inductive case, where $C = \mathsf{whdo}(H)(D)$: Let $y_1, \cdots, y_n$ be the list of all variables which occur in D and/or occur in H and/or occur freely in F or G. Let $z_1, \cdots, z_n$ be distinct variables, and disjoint from the y_i's. Define $L := \exists z_1 \cdots \exists z_n J$, where J is a formula defining the post relation for $(F, \mathsf{whdo}(K)(D))$, where

$$K := H \wedge (\neg y_1 \approx z_1 \vee \cdots \vee \neg y_n \approx z_n) \quad \text{(fortunately without quantifiers!)} .$$

The main property needed for L is that it is true for exactly those $(d_1, \cdots, d_n)$—values of $y_1, \cdots, y_n$ respectively—which satisfy the following property, which we shall refer to as $(*)$:

$(d_1, \cdots, d_n)$ arises as the output after D has been executed a finite (possibly zero) number of times, (1) starting with a state $(d'_1, \cdots, d'_n)$ where F is true; and (2) such that H is true immediately prior to each execution.

Let us assume this, and finish the main proof, then return to prove it.

We have two formulas and a F-H statement below which I claim are true in $\mathbf{N}$:

(i) $F \rightarrow L$: since, if F is true at $(d_1, \cdots, d_n)$, then so is L, by the case of zero executions in $(*)$.

(ii) $L \wedge \neg H \rightarrow G$: since, by $(*)$, a state where L is true and H is false is exactly a state arising after 'successfully' executing $\mathsf{whdo}(H)(D)$ starting from a state where F is true. But since $F\{\mathsf{whdo}(H)(D)\}G$ is here assumed to be true, it follows that G is true in such a state, as required.

(iii) $(L \wedge H)\{D\}L$: since, starting from a state where both L and H are true, executing D one more time as in $(*)$ still gives a state where L is true.

Now we can construct the required derivation of $F\{\mathsf{whdo}(H)(D)\}G$ as follows. By (iii) and the induction in the overall proof of this theorem, there is a derivation of $(L \wedge H)\{D\}L$. But we can then apply the two rules

$$\frac{(L \wedge H)\{D\}L}{L\{\mathsf{whdo}(H)(D)\}(L \wedge \neg H)}$$

$$\frac{F \rightarrow L \ , \ L\{\mathsf{whdo}(H)(D)\}(L \wedge \neg H) \ , \ (L \wedge \neg H) \rightarrow G}{F\{\mathsf{whdo}(H)(D)\}G}$$

to get the desired derivation. The second one displayed is indeed an instance of rule (IV) because of (i), (ii) and the fact that the oracle allows us to use any $(A \rightarrow B)$'s which are true in $\mathbf{N}$.

To actually complete the proof, here are the arguments both ways for the fact that $(*)$ holds exactly when L is true.

In one direction, let $\vec{d}$ be a state where $(*)$ holds. To show L is true in that state, let $\vec{d'}$ be a suitable input as in $(*)$. Now run $\mathsf{whdo}(K)(D)$ starting with state $(\vec{d'}, \vec{e}) = (\vec{d'}, \vec{d})$ for $(\vec{y}, \vec{z})$. This execution terminates with state $(\vec{d}, \vec{e}) = (\vec{d}, \vec{d})$ since the $\vec{e}$-component clearly doesn't change with D's execution, and, of course, $(\vec{d}, \vec{d})$ is a state where the

$$\text{“ } (\neg y_1 \approx z_1 \vee \cdots \vee \neg y_n \approx z_n)\text{-half of } K \text{ ”}$$

is false. Thus, $(\vec{d}, \vec{d})$ is a post state for $(F, \mathsf{whdo}(K)(D))$ since F holds for $(\vec{d'}, \vec{d})$, independently of the $\vec{d}$-half. Thus J is true at $(\vec{d}, \vec{d})$, and so L is true at $\vec{d}$, as required.

Conversely, suppose that L is true at $\vec{d}$, so that, for some $\vec{e}$, the formula J is true at $(\vec{d}, \vec{e})$. Thus there is a state $(\vec{d'}, \vec{e'})$ such that F is true at $\vec{d'}$ and executing $\mathsf{whdo}(K)(D))$ on $(\vec{d'}, \vec{e'})$ terminates with state $(\vec{d}, \vec{e})$. Therefore a finite number of executions of D exists, starting with $\vec{d'}$, where F is true at $\vec{d'}$, and terminating with $\vec{d}$. Also, for each state just prior to each execution, the formula K is true, and so H is also true. Thus $(*)$ holds for $\vec{d}$, as required.

Note how this proof of (relative) adequacy is in fact (relatively) effective, all relative to truth in $\mathbf{N}$ of course. Actually carrying out proofs of F-H statements is a matter of considerable interest, both by hand and mechanically.

8.2—Floyd-Hoare Logic for ATEN$_{\mathcal{L}}$.

Here we discuss generalizations of the theorems just proved. It is very nice that the F-H proof system is so simple, just one rule for each clause in the structural inductive definition of **ATEN**, plus one more rule to tie in with formal deductions in 1$^{\underline{\text{st}}}$order logic. Now, by the last (completeness) theorem, one expects to *prove* the following rules (which seem so fundamental that, a priori, one would have suspected that they (or something close) would need to be *part* of the basic system):

$$\frac{F\{C\}G\ ,\ F\{C\}H}{F\{C\}(G \wedge H)}$$

$$\frac{F\{C\}H\ ,\ G\{C\}H}{(F \vee G)\{C\}H}$$

However, after attempting these as an exercise, the reader may be even more disenchanted with this *oracular* F-H system. The only proofs I know would just go back to the inductive proof of **8.4** and keep pulling the oracle rabbit out of the hat. This hardly seems in the syntactic spirit of formal proof systems, much less in the spirit of something one would hope to automate.

But surely **8.4** has a good deal of syntactic content—we just need to generalize the situation of the last subsection to one where there are genuine axiomatic complete proof systems for truth in 1$^{\underline{\text{st}}}$order logic. One can do this merely by allowing *arbitrary* (in particular, *finite*) interpretations of the 1$^{\underline{\text{st}}}$order number theory language. But for later purposes, it is useful to discuss *arbitrary* 1$^{\underline{\text{st}}}$*order languages plus interpretations*, and the analogue of **ATEN** for them.

So let $\mathcal{L}$ be any 1$^{\underline{\text{st}}}$order language (with equality). Associated with it, one defines a command language **ATEN**$_{\mathcal{L}}$ in exact parallel to the original definition : The atomic commands are assignment commands using terms of the language. The inductive structure will use only whdo-commands and concatenation (;) for sequenced commands as before, with quantifier-free formulas (called "Boolean expressions" by CSers) from the language 'dictating the flow of control' in the former. (These are also used in ite$(H)(C)(D)$-commands, if the if-then-else command construction is added to the language, which is then often referred to as the 'while-language'—as we know, it's extra expressiveness in the informal sense does not increase the computational power, as long as the interpretation used

includes something equivalent to the set of all natural numbers.) For the semantics, one further fixes an interpretation of the language. Each of the usual "bins" contains an element of the interpretation. Then one can talk about F-H statements as before, with the so-called pre- and post- conditions being arbitrary formulas from the language. (Note that in [**C**], for example, the situation is generalized a bit further, with *two* $1^{\underline{\text{st}}}$order languages, one for each of the above uses, one being a sublanguage of the other. This seems not to be a major generalization, and will be avoided here for pedagogical reasons. Of course [**C**] also involves a more complicated command language, very similar to the one discussed in the next subsection.)

Now we note that, whatever the language, for any interpretation which is a *finite* set, truth is actually decidable, so there will certainly be a rather straightforward axiomatic complete proof system. (Expressed alternatively, the set of true formulas with respect to such an interpretation is a decidable set, so it's certainly a recursively enumerable set.) And so, the following theorem, whose proof is essentially the same as that of **8.4**, does have plenty of relevant examples.

To be convincing about that, here is the argument that $(\mathcal{L}, I)$ is expressive, as defined in the theorem just below, as long as I is a finite interpretation of $\mathcal{L}$. Let $x_1, \cdots, x_n$ be the variables occurring in C and/or free in F. Let $E \subset I^n$ be the set of values of those variables where $Q_{F,C}$ is true; that is outputs from C executed on some input where F is true. Now define G to be

$$\vee_{(a_1,\cdots,a_n)\in E}(x_1 \approx a_1 \ \wedge \cdots \wedge x_n \approx a_n) \quad .$$

This clearly does the job (and makes sense since E is a finite set!)

Theorem 8.5. *Assume the pair $\mathcal{L}, I$ is* expressive *: for any F and C there is a formula $G_{F,C}$ which is true at exactly those states which arise as the output from C applied to an input state where F is true.*

Then the combined F-H proof system for $\mathbf{ATEN}_{\mathcal{L}}$*, using the original set of four rules, plus an oracle, as discussed earlier, for (the $1^{\underline{\text{st}}}$order language $\mathcal{L}$ plus its given interpretation I), is itself* complete*, in that every F-H statement which is true in the given interpretation can be derived using that combined system.*

8.3—F-H Logic for more complicated (and realistic?) languages.

The next job is to describe a more 'realistic' version of this theorem, basically Cook's original theorem. Because the command language defined below includes procedure calls and variable declarations, the formulation of the system, and the proof of its soundness and adequacy, all require considerable care.

Now this language will not allow recursive programming, and its semantics uses what is called *dyamic scope*, making even it fairly unrealistic as an actual language in which to write lots of programs. The choice of Cook's language was made partly because of the apparent lack of a complete, painstakingly-careful treatment in the literature. Giving a smoother development would have been, say, a language allowing only *parameterless* procedures, even including recursive programming. Such a language is dealt with briefly in the third addendum below, and in reasonable detail in [**Apt**] (and also in [**deB**] except for disallowing nested procedures).

Cook refers somewhat unspecifically to instances in the literature of F-H systems which turned out inadvertantly to **not** be sound, including earlier versions of the system in [**C**]. The reader might like to look at four quotations from papers on the subject, easily found in the final subsection, before the addenda. These will help brace and motivate you for the painstaking technicalities at times here, though at first reading, maybe skipping a lot of those technicalities is desirable. On the other hand, they explain the need for the caveats given later which circumscribe the language **DTEN** we start with, producing a sublanguage **CTEN**. The language **DTEN** likely admits no complete F-H proof system, but the semantics given below are also probably 'wrong' anyway, without the restrictions to **CTEN**, or something similar. Giving rigorous semantics for complicated command languages has been the subject of some controversy over the years, including how necessary the use of denotational semantics is for this. At any rate, it seems to be only at the point where one has procedures both with parameters and allowing recursive programming that denotational semantics becomes particularly useful (or else if one insists on having GOTO commands, or declarations of functions, etc.) None of these three possibilities will be considered in this write-up. As mentioned above, in the third addendum, we do consider briefly a language with recursive programming, and give an F-H proof system for it, but, for that one, parameters are avoided. Function declarations appear briefly in the first addendum.

Before launching into this, it is worth mentioning a couple of more specific things about earlier instances of technical errors. As indicated below in the middle of heavy technicalities, we do fix one technical error in [**C**] which seems not to have been noticed before. But the statements of the main results of general interest in [**C**] are all correct, modulo the discussion in the next paragraph.

In [**C+**], there is a different correction to [**C**], related to the treatment of the declaration of variables. The two methods given there for fixing this seem a bit ad hoc, so below we have defined the semantics of the language a little differently than is done elsewhere. Rather than just updates to the 'store' or 'contents of the bins', we add also an update to

the function which associates variables to bins. So the semantics of a given command plus input, when defined, is not just a sequence of bin contents, but each term in the sequence has a second component, mapping variables injectively to bins. Actually, the only type of command which changes the latter is the variable declaration block command. This modification of the semantics seems to me a considerably more natural way to do things, and it does take care of the small problem in Cook's original published paper [**C**] for which [**C+**] gave the other fixes. Compare the definitions of *Comp* in [**C**] and [**Apt**] with the definition of *SSQ* below, to see more precisely what is being referred to here.

The two languages considered in this subsection are essentially fragments of ALGOL. We'll define **DTEN** by structural induction, but with a slight complication compared to earlier inductive definitions. There will be both a set of commands and a set of declarations. These are defined simultaneously by a mutual recursive method.

I suspect that a major reason for the popularity of BNF-notation in the CS literature (introduced more-or-less simultaneously with ALGOL around 1960) is that it largely obviates the need for conscious cogitation concerning this sort of structural inductive definition! The language **DTEN** has the "**D**" for "declaration" perhaps. The language **CTEN** is singled out from **DTEN** by a list of restrictions ('caveats') much later, and has "**C**" for caveat, or for Cook perhaps. Ours will be slightly simpler than Cook's language, but only by dropping the **if-thendo-elsedo**-command from **BTEN**, which is redundant in a sense, as mentioned in the last subsection. Dropping it just gets rid of an easy case in several lengthy definitions and proofs. A reader who wants that command constructor can easily add it in, along with its semantics, figure out what the extra rule of inference would need to be (see also Addendum 3 here), and then prove completeness for herself, checking her work against [**C**] if desired.

Definition of the command language.

We define two string sets, COM, of commands, and DEC, of declarations, by simultaneous structural induction. This will depend on a fixed $1^{\underline{\text{st}}}$order language whose name will be suppressed from the notation. The *command language* will of course be independent of which interpretation is used. (But the *semantics* of the command language will depend on the interpretation used for the $1^{\underline{\text{st}}}$order [assertion] language.) We'll define the whole thing below in the same style (a bastardization of BNF-notation) which was used earlier for the denotational semantics of **ATEN**. Recall that X^* is the set of finite strings of elements from some set X. Also we let $X^{*!}$ be the set

of finite strings of *distinct* elements from X.

$$x \in IDE\ ;\ p \in PRIDE\ ;\ \vec{u}, (\vec{x} : \vec{v}) \in IDE^{*!}\ ;\ t \in TRM\ ;\ \vec{t} \in TRM^*\ ;\ H \in FRM_{\text{free}}$$

where TRM is the set of terms from the $1^{\underline{\text{st}}}$order language, FRM_{free} its set of quantifier-free formulas, IDE is its set of variables (or *identifiers*), and $PRIDE$ is a set of identifiers for 'procedures', which might as well be disjoint from IDE. When paired as $(\vec{u}, \vec{t})$ below, we always require that "$\vec{u}$" be disjoint from "$\vec{t}$". The notation "$\vec{t}$" simply means the set of all variables which appear in the various terms t_i. That notation occurs often, to save words.

Here is the **Definition of DTEN**:

$$D, D_1, \cdots, D_k \in DEC\ :=\ \{\ \mathsf{new}\ x\ \ ||\ \ \mathsf{proc}\ p(\vec{x} : \vec{v}) \equiv C\ \mathsf{corp}\ \}$$

$$C, C_1, \cdots, C_n \in COM\ :=\ \{\ x \doteqdot t \mid \mathsf{call}\ p(\vec{u} : \vec{t})\ ||\ \mathsf{whdo}(H)(C)\ \mid \mathsf{begin}\ D_1; \cdots; D_k\ ;\ C_1; \cdots; C_n\ \mathsf{end}\ \}$$

The last construction allows k and/or n to be 0. And so, in inductive constructions and proofs for **DTEN** or **CTEN**, there will tend to be seven cases: the assignment and call (atomic) commands, the whdo-command, and then the cases $k = 0 = n$ (do-nothing command), $k = 0 < n$ (no declarations), and $k > 0$ (two cases, depending on which type of declaration is the leftmost, D_1, in the list). Actually, our distinction between declarations and commands is a bit moot, since the case $k = 1$ and $n = 0$ gives begin D end as a command for any declaration D.

Notice that a call subcommand can occur without any corresponding procedure declaration within the command; in fact the call command on its own is such a command. This is necessary in order to have inductive constructions and proofs, even if it would seldom occur in a useful program. Thus, below in the semantics, we need an extra component, π, which (sometimes) tells us which procedure (another command) to use when p is called. The object π itself is modified by the semantics to conform with whatever procedure declarations occur within the command being 'semantified'. Especially in the proof system, a version of π becomes more part of the syntax.

The functions s and δ, the other two extra semantic components besides π and the given interpretation are explained and used below. They are needed in order to make a separation between variables and the bins/store/memory,

which can occur in more practical ALGOL-like languages (as opposed to **ATEN/BTEN**, where no distinction is made). Treatments bypassing the bins, e.g. **[Cl1], [Old1]**, can be elegant and useful, but harder to motivate.

We need to modify our notation in **[LM]** for the semantics of $1^{\underline{\text{st}}}$order languages to conform with the above. How to do this will be fairly obvious. We consider functions m from a finite subset of identifiers (or variables) to the underlying set, I, of the interpretation. Further down m will always be $s \circ \delta$, so we frequently consider the 'value' $t^{s\circ\delta} \in I$, for a term t; and whether a formula F is true or not "at $s \circ \delta$".

N.B. "F is true at $s|\delta$" will mean exactly the same thing as "F is true at $s \circ \delta$". This will be convenient notation. But elsewhere, $s|\delta$ is just a convenient way to denote the ordered pair (s, δ). It is different than the function $s \circ \delta$.

To be semi-precise, the element t^m of I is undefined unless all variables in t are in the domain of m; and otherwise, it is what we used to call $t^{\underline{v}}$, when the identifiers are the variables x_i, and where the ith component, v_i, of $\underline{v}$ is $m(x_i)$ if that is defined; and v_i is uninteresting for the majority of i, those with x_i not in the domain of m.

Similarly, a formula being true or false at m can only hold when all free variables in the formula are in the domain of m, and then, at least when the identifiers are the variables x_i, it means the same as the formula being true at $\underline{v}$, which is related to m as in the paragraph above.

Next we discuss the objects π mentioned above. These will be sometimes written as functions

$$PRIDE \supset A \xrightarrow{\pi} COM \times IDE^{*!} \times IDE^{*!} \subset COM \times IDE^{*!} ,$$

so $\pi(p)$ has the form $(K, (\vec{x} : \vec{v}))$, where K is a command, and $(\vec{x} : \vec{v})$ is a list of distinct variables. The colon-divider between the 'x-part' and the 'v-part' will be later used to indicate for example that the v-part is not allowed to appear on the left-hand side of an assignment command, because we will substitute terms that might not be variables for it. The function π is defined for procedure identifiers p from a given **finite** subset, A, of $PRIDE$. This will say which procedure K in COM to activate when p is called. But, as we see below, in the definition of SSQ for the command

$$\mathsf{begin\ proc}\ p(\vec{x} : \vec{v}) \equiv K\ \mathsf{corp}\ ;\ D_1; \cdots ; D_k\ ;\ C_1; \cdots ; C_n\ \mathsf{end}\ ,$$

the value $\pi(p)$ will have been 'changed' to the correct thing if a procedure declaration for the name p has already occurred within the same begin—end-'block' before the call takes place. Interchangeably, we can write

$$\pi(p) \;=\; [\mathsf{proc}\ p(\vec{x} : \vec{v}) \equiv K\ \mathsf{corp}]\ .$$

Since π has been introduced somewhat as a semantic concept, and we wish to use it also syntactically, here is another alternative notation. Just as F, C and G are names for strings of 'meaningless' symbols, so too we shall take " / π " as short for a string

$$/\ p_1(\vec{x_1} : \vec{v_1}) \equiv K_1\ /\ p_2(\vec{x_2} : \vec{v_2}) \equiv K_2\ /\ \cdots\ /\ p_n(\vec{x_n} : \vec{v_n}) \equiv K_n\ \ .$$

We shall have given $PRIDE$ a fixed total order $\prec$, and the finite set $\{p_1 \prec p_2 \prec \cdots \prec p_n\} \subset PRIDE$ will be the domain of π with the inherited order, with $\pi(p_i) \;=\; (K_i, (\vec{x_i} : \vec{v_i}))$.

We shall also make minor use later of a total order on IDE, so assume that one is given to begin.

Though it seems not to appear in exactly this form in the literature, a more fundamental syntactic concept than a command C in a language with procedure calls seems to be a string C/π, where π is the sort of string just above. So these may still be conceived as syntactically generated strings of meaningless symbols. Allowing call-commands on their own, and introducing the $/\pi$-notation, are very convenient for theoretical considerations. An actual program for computation would presumably have a declaration for any call, and so "$/\pi$" would be redundant, and then we can think of it as the empty string (or function with empty domain) for such a program.

Definition of DTEN's semantics.

A preliminary definition needed is for the *'simultaneous' substitution* of terms for *free* variables in commands, as used below in defining SSQ of a call-command with $K^{[\ \vec{x}\to\vec{u}\ ,\ \vec{v}\to\vec{t}\ ]}$. As the author of **[C]** knows perfectly well (though some of his readers need lots of patience to see), defining, as on his pp.94-95, substitution in terms of free variables, then free variables in terms of substitution, is not circular, but can be made as a recursive definition in a sense. But this takes more than a tiny amount of thought, when, as in that paper, the data given by π is not made explicit. Another point motivating a more explicit approach is that substitution cannot always be done in a command; for example, substitution of a term which is *not* a variable for a variable x in a command which includes an assignment to x.

So we'll first define substitution for commands C by structural induction, then for strings π by induction on the length of the string. Let $\vec{z} = (z_1, \cdots, z_n)$ and $\vec{e} = (e_1, \cdots, e_n)$ be finite strings (of the same length) of variables z_i and terms e_i, with the variables distinct.

Simultaneous with the inductive definition below, we need the trivial proof by induction of the fact that $(\mathsf{begin}\ D_*; C_*\ \mathsf{end})^{[\vec{z}\to\vec{e}\ ]}$, if defined, always has the form $\mathsf{begin}\ D'_*; C'_*\ \mathsf{end}$, where the sequences D'_*, C'_* have the same lengths as D_*, C_* respectively.

The substitution $(x \div t)^{[\vec{z}\to\vec{e}\ ]}$ is defined when $x = z_i \Rightarrow e_i$ is a variable, and the answer is $x^{[\vec{z}\to\vec{e}\ ]} \div\ t^{[\vec{z}\to\vec{e}\ ]}$. The substitutions are the obvious substitution into terms as defined in **[CM]**.

The substitution $(\mathsf{call}\ p(\vec{u} : \vec{t}))^{[\vec{z}\to\vec{e}\ ]}$ is not defined when
$(\exists i, j,\ u_i = z_j$ and e_j is not a variable), and also when it would produce repeated variables left of the colon or the same variable occurring on both sides of the colon. When it is defined, the answer is

$$\mathsf{call}\ p(\vec{u}^{\ [\vec{z}\to\vec{e}\ ]} : \vec{t}^{\ [\vec{z}\to\vec{e}\ ]}) \ .$$

This notation means : "do the substitution in each of the terms in both strings".

Define

$$(\mathsf{whdo}(H)(C))^{[\vec{z}\to\vec{e}\ ]} \ :=\ \mathsf{whdo}(H^{[\vec{z}\to\vec{e}\ ]})(C^{[\vec{z}\to\vec{e}\ ]})$$

where substitution into formulas is as defined in **[CM]**.

Define

$$(\mathsf{begin\ \ end})^{[\vec{z}\to\vec{e}\,]} \ := \ \mathsf{begin\ \ end}\ .$$

Define

$$(\mathsf{begin}\ C_1; C_*\ \mathsf{end})^{[\vec{z}\to\vec{e}\,]} \ := \ \mathsf{begin}\ C_1^{[\vec{z}\to\vec{e}\,]}; C'_*\ \mathsf{end}\ ,$$

where

$$(\mathsf{begin}\ C_*\ \mathsf{end})^{[\vec{z}\to\vec{e}\,]} \ = \ \mathsf{begin}\ C'_*\ \mathsf{end}\ .$$

To be straightforward about this last one

$$(\mathsf{begin}\ C_1; \cdots; C_k\ \mathsf{end})^{[\vec{z}\to\vec{e}\,]} \qquad \text{is simply} \qquad \mathsf{begin}\ C_1^{[\vec{z}\to\vec{e}\,]}; \cdots; C_k^{[\vec{z}\to\vec{e}\,]}\ \mathsf{end}\ .$$

Define

$$(\mathsf{begin\ new}\ x\ ;\ D_*\ ;\ C_*\ \mathsf{end})^{[\vec{z}\to\vec{e}\,]} \ := \ \mathsf{begin\ new}\ x'\ ;\ D'_*\ ;\ C'_*\ \mathsf{end}\ ,$$

where we let x' be the first variable 'beyond' all variables occurring in the string (including $\vec{z}$ and $\vec{e}$), then let D''_* and C''_* be obtained from D_* and C_* respectively by replacing all occurrences of x by x', and then define the right-hand side in the display by (using the induction on length)

$$\mathsf{begin}\ D'_*; C'_*\ \mathsf{end} \ := \ (\mathsf{begin}\ D''_*; C''_*\ \mathsf{end})^{[\vec{z}\to\vec{e}\,]}\ .$$

The reason that x is first replaced is because some of the e_i may have occurrences of x, and we don't want those x's to be governed by the "new" when they appear after substitution for the z's corresponding to those e's. A similar remark applies in the final case below. (One begins to appreciate why CSers tend to want to have substitution always defined, and tend to give a correspondingly involved definition when doing basic logic and λ-calculus. And we do need to fix a linear order beforehand on the set IDE, to make sense above of the "...first variable 'beyond' all variables ...".)

Finally define

$$(\mathsf{begin\ proc}\ p(\vec{x} : \vec{v}) \equiv K\ \mathsf{corp}\ ;\ D_*\ ;\ C_*\ \mathsf{end})^{[\vec{z}\to\vec{e}\,]} \ :=$$

$$\mathsf{begin\ proc}\ p(\vec{x}' : \vec{v}') \equiv K'\ \mathsf{corp}\ ;\ D'_*\ ;\ C'_*\ \mathsf{end}\ ,$$

where we let $\vec{x}'$ and $\vec{v}'$ be the first strings of distinct variables beyond all variables occurring in the string, then let K', D''_* and C''_* be obtained from K, D_* and C_* respectively by replacing all occurrences of x_i by x'_i and v_j by v'_j, and finally define the rest of the right-hand side in the display by

$$\mathsf{begin}\ D'_*; C'_*\ \mathsf{end}\ :=\ (\mathsf{begin}\ D''_*; C''_*\ \mathsf{end})^{[\vec{z}\to\vec{e}\ ]}\ .$$

For substituting into π itself, define

$$(\ p_1(\vec{x}_1 : \vec{v}_1) \equiv C_1\ /\ p_2(\vec{x}_2 : \vec{v}_2) \equiv C_2\ /\ \cdots\ /\ p_n(\vec{x}_n : \vec{v}_n) \equiv C_n)^{[\vec{z}\to\vec{e}\ ]}\ :=$$

$$(\ p_1(\vec{x}_1 : \vec{v}_1) \equiv C_1\ /\ \cdots\ /\ p_{n-1}(\vec{x}_{n-1} : \vec{v}_{n-1}) \equiv C_{n-1})^{[\vec{z}\to\vec{e}\ ]}\ /\ p_n(\vec{x_n} : \vec{v_n}) \equiv C_n^{[\vec{z'}\to\vec{e'}\ ]}\ ,$$

with $\vec{z'}$ being $\vec{z}$ with all the variables in "$\vec{x}_n$" $\cup$ "$\vec{v}_n$" removed, and $\vec{e'}$ being the terms in $\vec{e}$ corresponding to $\vec{z'}$. Start the induction in the obvious way, with $\emptyset^{[\vec{z}\to\vec{e}\ ]}\ :=\ \emptyset$, the empty string.

The definition of *free variable* or *global variable* in a command is itself inductive, and really it applies not to just a command, but a pair, *command* / π. (We can always take $\pi = \emptyset$.) The induction is on the cardinality of the domain of π, and then, for fixed such cardinality, structural induction on C. (This obviates the need to discuss "substituting the body of p for each call to p, over-and-over till no calls remain", which will only work because we shall be excluding recursive commands.)

$FREE(x \doteqdot t\ /\ \pi)$ consists of x and all variables occurring in t.

$FREE(\mathsf{call}\ p(\vec{u} : \vec{t})\ /\ \pi)$ consists of the u_i, the variables in the t_j, and,

(i) if $p \notin \mathrm{dom}(\pi)$, or if $p \in \mathrm{dom}(\pi)$ with $\pi(p) = (K, (\vec{x} : \vec{v}))$ but $K^{[\ \vec{x}\to\vec{u}\ ,\ \vec{v}\to\vec{t}\ ]}$ does not exist, then nothing else; whereas

(ii) if $p \in \mathrm{dom}(\pi)$ with $\pi(p) = (K, (\vec{x} : \vec{v}))$, and $K^{[\ \vec{x}\to\vec{u}\ ,\ \vec{v}\to\vec{t}\ ]}$ does exist, then include also the set $FREE(K^{[\ \vec{x}\to\vec{u}\ ,\ \vec{v}\to\vec{t}\ ]}\ /\ \pi')$, where π' is π, except that p is removed from its domain. (This is the only of the seven cases where the inductive hypothesis on the cardinality of π is used. N.B. We shall later be excluding recursive commands, and the use of π' rather than π here and later is justified by that.)

$FREE(\mathsf{whdo}(H)(C)\ /\ \pi)$ consists of $FREE(C/\pi)$ and the variables in H. (Of course, H is quantifier-free.)

$FREE(\mathsf{begin\ \ end}\ /\ \pi)$ is empty.

$FREE(\mathsf{begin}\ C_*\ \mathsf{end}\ /\ \pi)$ is the union of the $FREE(C_i/\pi)$—more properly, inductively

$$FREE(\mathsf{begin}\ C_1; C_*\ \mathsf{end}\ /\ \pi)\ :=\ FREE(C_1/\pi) \cup FREE(\mathsf{begin}\ C_*\ \mathsf{end}\ /\ \pi)\ .$$

$$\text{Define}\quad FREE(\mathsf{begin\ new}\ x\ ;\ D_*\ ;\ C_*\ \mathsf{end}\ /\ \pi)\ :=$$
$$FREE(\mathsf{begin}\ D_*\ ;\ C_*\ \mathsf{end}\ /\ \pi) \setminus \{x\}\ ;$$

that is, 'remove' the variable x .

$$\text{Define}\quad FREE(\mathsf{begin\ proc}\ p(\vec{x} : \vec{v}) \equiv K\ \mathsf{corp}\ ;\ D_*\ ;\ C_*\ \mathsf{end}\ /\ \pi)\ :=$$
$$[FREE(K/\pi) \setminus (\text{“}\vec{x}\text{ ”} \cup \text{“}\vec{v}\text{ ”})]\ \cup\ FREE(\mathsf{begin}\ D_*\ ;\ C_*\ \mathsf{end}\ /\ \pi)\ .$$

As indicated earlier, we shall use here (and many times below) notation such as “$\vec{x}$ ” ∪ “$\vec{v}$ ” and just “$\vec{x}$ ” in the obvious way, as the underlying set of variables in the list.

For the 'free' or 'so-called 'global' variables in a command C itself, just take π to be empty; that is, the set is $FREE(C/\emptyset)$.

The real 'meat' of the semantics will be a function $SSQ(C/\pi, s|\delta)$ of two (or three or four) 'variables', the 'State $Se$$Q$uence', as defined below. We shall define, inductively on n, the nth term, $SSQ_n(C/\pi, s|\delta)$ of that sequence, by structural induction on the command C. So it's a form of double induction.

The other two input components for SSQ are as follows, where we assume that an interpretation of our underlying $1^{\underline{\text{st}}}$order language has been fixed once-and-for-all.

We need an **injective** function

$$IDE\ \supset\ B\ \xrightarrow{\ \delta\ }\ \{\mathrm{bin}_i \mid i \geq 0\ \}\ ,$$

so δ 'locates' the variables from a **finite** subset B of variables as having values from suitable bins. Or, as we shall sometimes say, it 'associates' some variables with bins.

And we need

$$s \ : \ \{\mathrm{bin}_i \mid i \geq 0 \} \ \longrightarrow \ I \ ,$$

so s is the 'state', telling us which element of the interpretation I (which 'value') happens to be 'in' each bin. Sometimes the word "state" would rather be used for the composition $s \circ \delta$, or for the pair $s|\delta$.

Definition of the computational sequence SSQ.

The value, $SSQ(C/\pi, s|\delta)$, of the semantic function SSQ will be a non-empty (possibly infinite) sequence $\prec s|\delta \ , \ s'|\delta' \ , \ s''|\delta'' \ , \ \cdots \succ$ of pairs, which is what we envision as happening, step-by-step, to the bin contents and variable associations with bins, when the command C is executed beginning with 'environment' $[s|\delta, \pi]$. Note, from the definition below, that the second, 'δ', component remains constant in the semantics of any command which has no variable declaration block. The domain of SSQ is not *all* quadruples— as we see below, if C includes a call to p and $p \notin \mathrm{dom}(\pi)$, then $SSQ(C/\pi, s|\delta)$ *might* not be defined, but it normally will be if the call is within a block, begin $\cdots$ end, which earlier includes a declaration of p. Furthermore, if C includes an assignment command involving a variable on which δ is not defined, again SSQ might not be defined.

The seven cases above in defining freeness (and substitution) will occur in exactly that same order, each of the three times, immediately below in twice defining SSQ.

First we give the definition of SSQ_n in uncompromising detail; many can skip this and go to the second form of the definition just afterwards. When abort occurs below, that means that SSQ in that case is undefined. Whereas, if non-existent occurs, that means the term of the sequence being considered is undefined because the sequence is too short. Also we let

$$OUT(C/\pi, s|\delta) := \begin{cases} \text{last term in } SSQ(C/\pi, s|\delta) & \text{if the latter is a finite sequence;} \\ \text{indifferent} & \text{otherwise.} \end{cases}$$

Here is the doubly inductive definition of $SSQ_n(C/\pi, s|\delta)$:

When $n = 0$, the seven cases, in order, define $SSQ_0(C/\pi, s|\delta)$ to be, respectively:

(1) abort if some variable in t is not in the domain of δ ; otherwise
$[\mathrm{bin}_i \ \mapsto \ \mathsf{if}\ \delta(x) = \mathrm{bin}_i\ , \mathsf{then}\ t^{s\circ\delta}, \mathsf{else}\ s(\mathrm{bin}_i)\ ] \mid \delta$;

(2) abort if $p \notin \mathrm{dom}(\pi)$, or if $p \in \mathrm{dom}(\pi)$ with $\pi(p) = (K, (\vec{x} : \vec{v}))$ and $K^{[\ \vec{x}\to\vec{u}\ ,\ \vec{v}\to\vec{t}\ ]}$ does not exist ; otherwise $s|\delta$,

(3) abort if some variable in H is not in the domain of δ; otherwise $SSQ_0(C'/\pi, s|\delta)$ or $s|\delta$ depending on whether H is true or false at $s \circ \delta$;

(4) $s|\delta$,

(5) $SSQ_0(C_1/\pi, s|\delta)$,

(6) $s|\delta$,

(7) $s|\delta$.

For the inductive step on n, the seven cases, in order,
define $SSQ_{n+1}(C/\pi, s|\delta)$ to be, respectively:

(1) non-existent;

(2) abort if $p \notin \mathrm{dom}(\pi)$, or if $p \in \mathrm{dom}(\pi)$ with $\pi(p) = (K, (\vec{x} : \vec{v}))$ and $K^{[\ \vec{x}\to\vec{u}\ ,\ \vec{v}\to\vec{t}\ ]}$ does not exist ; otherwise
$SSQ_n(K^{[\ \vec{x}\to\vec{u}\ ,\ \vec{v}\to\vec{t}\ ]}\ /\pi, s|\delta)$ (which of course might itself be undefined) .

(3) abort, if $\delta(y)$ is not defined for some variable y occurring in H; otherwise non-existent, if H is false at $s \circ \delta$; otherwise
$SSQ_{n+1}(C'/\pi, s|\delta)$, if the latter is defined and H is true at $s \circ \delta$; otherwise
$SSQ_{n-k}(\mathsf{whdo}(H)(C')/\pi\ ,\ OUT(C'/\pi, s|\delta))$, if $SSQ(C'/\pi, s|\delta)$ has length $k+1$ with $k \le n$ and H is true at both $s|\delta$ and $OUT(C'/\pi, s|\delta)$.

(4) non-existent,

(5) $SSQ_{n+1}(C_1\ /\pi, s|\delta)$, if $SSQ(C_1\ /\pi, s|\delta)$ has length $k+1$ with $k > n$ (i.e. $SSQ_0, \cdots SSQ_k$ all exist but no others);
$SSQ_{n-k}(\mathsf{begin}\ C_*\ \mathsf{end}\ /\pi, OUT(C_1/\pi, s|\delta))$, if $SSQ(C_1/\pi, s|\delta)$ has length $k+1 \le n+1$ [so $OUT(C_1/\pi, s|\delta) = SSQ_k(C_1/\pi, s|\delta)$];

(6) Define

$$m \ = \ \min\{\ a \ : \ \ \delta(z) = \mathrm{bin}_b \implies b < a\ \}\ ;$$

$$\delta' \ := \ (y \mapsto [\textsf{if } y = x \ \textsf{ then } \mathrm{bin}_m \ \textsf{ else } \delta(y)]) \ ;$$

$$LOUT \text{ as the lefthand component of } OUT \ ;$$

and

$\delta''(z) \ := \ \mathrm{bin}_m$ if z is the first variable after $\mathrm{dom}(\delta)$, but $\delta'' = \delta$ otherwise ;

so $\mathrm{dom}(\delta'') = \mathrm{dom}(\delta) \cup \{z\}$. (This uses the linear order on IDE.)

Then the answer is :
non-existent if $SSQ(\textsf{begin } D_* \ ; \ C_* \ \textsf{end}/\pi \ , \ s|\delta')$ has length less than n ;
$SSQ_n(\textsf{begin } D_* \ ; \ C_* \ \textsf{end}/\pi \ , \ s|\delta')$, if it has length greater than n ;
and
$LOUT(\textsf{begin } D_* \ ; \ C_* \ \textsf{end}/\pi \ , \ s|\delta') \mid \delta''$ if the length is n ; i.e. if
$OUT(\textsf{begin } D_* \ ; \ C_* \ \textsf{end}/\pi \ , \ s|\delta') = SSQ_{n-1}(\textsf{begin } D_* \ ; \ C_* \ \textsf{end}/\pi \ , \ s|\delta')$.

(7) $SSQ_n(\textsf{begin } D_* \ ; \ C_* \ \textsf{end}/\pi' \ , \ s|\delta)$, where π' is defined by

$$q \mapsto [\textsf{if } q = p \ \textsf{ then } (K, (\vec{x} : \vec{v})) \ \textsf{ else } \pi(q)] \ .$$

In 'delicate' cases later in the technical proofs, always refer back, if necessary, to this definition above; but now let us repeat the definition in a more readable form, but one where the 2nd and 3rd cases appear to be circular at first glance, because of the hidden induction on n. We have included a more 'CSer style' explanation, which may help neophytes reading similar things in the CS literature. So here's the definition of the sequence $SSQ(C/\pi, s|\delta)$ as a whole, with the seven cases in the usual order:

(1) $\underline{C \ = \ x \doteqdot t}$: The answer is

$$\prec s' \ : \ \mathrm{bin}_i \ \mapsto [\ \textsf{if } \delta(x) = \mathrm{bin}_i \ , \textsf{then } t^{s \circ \delta}, \textsf{else } s(\mathrm{bin}_i) \ ] \mid \delta \succ$$

(a sequence of length 1), and so $SSQ(\ x \doteqdot t \ / \ \pi, s|\delta)$ is undefined exactly when y is not in $\mathrm{dom}(\delta)$ for some variable y which occurs in the term t.

(2) $\underline{C \ = \ \textsf{call } p(\vec{u} : \vec{t} \ )}$: Here $SSQ(C/\pi, s|\delta)$ is undefined unless the left-hand side just following and $K^{[\ \vec{x} \to \vec{u} \ , \ \vec{v} \to \vec{t} \ ]}$ are both defined. Suppose that

$\pi(p) = (K, (\vec{x} : \vec{v}))$. Then the answer is:

$$\prec \ s|\delta \ , \ SSQ(K^{[\ \vec{x}\to\vec{u}\ ,\ \vec{v}\to\vec{t}\ ]}/\pi, s|\delta) \ \succ \ ,$$

where $\prec A, B \succ$ for two sequences means the first followed by the second, as long as A is finite, but just means A if the latter is an infinite sequence. So $SSQ(\ \mathsf{call}\ p(\vec{u} : \vec{t}\ )/\pi, s|\delta)$ is undefined exactly when p is not in $\mathrm{dom}(\pi)$ or $SSQ(K^{[\ \vec{x}\to\vec{u}\ ,\ \vec{v}\to\vec{t}\ ]}\ /\ \pi, s|\delta)$ is undefined (though that is a highly 'unoperational' criterion—see just below). The case that $K^{[\ \vec{x}\to\vec{u}\ ,\ \vec{v}\to\vec{t}\ ]}$ itself is undefined (perhaps because t_i is not merely a variable for some i such that v_i has an occurrence 'to the left of a colon') will not happen later, because the restrictions defining **CTEN** will forbid that possibility.

A remark is needed here. The command K might very well itself contain some call-subcommands (in fact, even perhaps calling p itself, though that would be excluded in the sublanguage **CTEN** carved out of **DTEN** later). In any case, this seems to destroy the inductive nature of this definition. (A similar remark occurs below for whdo.) But we are simply generating a sequence, by syntactic substitution ('call-by-name' if you like). So the definition above will certainly generate a unique sequence, though one which might be infinite. (Our previous pedantic definition, 'term-by-term', makes this explanation unnecessary.) The most extreme example of this would be where K itself is $\mathsf{call}\ p(\vec{x} : \vec{v}\ )$ and where we take $(\vec{u} : \vec{t}\ )$ to be $(\vec{x} : \vec{v}\ )$. There, the SSQ sequence S is given by the definition as $S = \prec s|\delta, S \succ$. It is a trivial induction on n to show that the nth term of any sequence S satisfying this must always be $s|\delta$ (and obviously S must be an infinite sequence). So S has been defined as the (expected) constant infinite sequence.

Now we resume giving the inductive definition of $SSQ(C/\pi, s|\delta)$.

(3) $\underline{C \ = \ \mathsf{whdo}(H)(C')}$: Recall that

$$OUT(C'/\pi, s|\delta) := \begin{cases} \text{last term in } SSQ(C'/\pi, s|\delta) & \text{if the latter is a finite sequence;} \\ \text{indifferent} & \text{otherwise.} \end{cases}$$

Then the answer is undefined if some variable in H is not in the domain of δ ; and otherwise is :

if H is false at $s \circ \delta$,then $\prec s|\delta \succ$, else

$$\prec SSQ(C'/\pi, s|\delta) \ , \ SSQ(\mathsf{whdo}(H)(C')/\pi, \ OUT(C'/\pi, s|\delta)) \succ \quad .$$

This is not a fixed point equation, in any subtle sense, despite the (non-appearing) left-hand side appearing more-or-less as part of the right-hand side. The given definition generates a state sequence, possibly infinite, in a perfectly straightforward manner. Here it is true that, when $SSQ(C'/\pi, s|\delta)$ is undefined and H is true at $s \circ \delta$, then $SSQ(\ \mathsf{whdo}(H)(C')/\pi, s|\delta)$ is undefined. But $SSQ(\ \mathsf{whdo}(H)(C')/\pi, s|\delta)$ may also be undefined for reasons related to repeated execution of C' possibly 'changing' δ and/or π.

(4) $\underline{C \ = \ \mathsf{begin \ \ end}}$: The answer is $\quad \prec s|\delta \succ \quad .$

(5) $\underline{C \ = \ \mathsf{begin}\ C_1; C_*\ \mathsf{end}}$: where C_* means $C_2; \cdots ; C_k$, including the case $k = 1$ where the latter is blank. Here the answer is:

$$\prec SSQ(C_1/\pi, s|\delta) \ , \ SSQ(\mathsf{begin}\ C_*\ \mathsf{end}/\pi, \ OUT(C_1/\pi, s|\delta)) \succ \ .$$

(6) $\underline{C \ = \ \mathsf{begin\ new}\ x \ ; \ D_* \ ; \ C_*\ \mathsf{end}}$: With m, δ', δ'' and $LOUT$ defined as in the more pedantic version of the definition of SSQ, the answer is:

$$\prec \ s|\delta \ , \ SSQ(\mathsf{begin}D_*; C_*\mathsf{end}/\pi, s|\delta') \ , \ LOUT(\mathsf{begin}D_*; C_*\mathsf{end}/\pi, s|\delta') \mid \delta'' \ \succ$$

The intended effect here is that, when executing this block, the variable x begins with whatever value is in the first of the ultimate consecutive string of 'unassigned to variables' bins, that is, the mth bin (presumably 'placed there as input'). But it is restored to the value, if any, it had before entry to the block, immediately upon exit from the block. Because of the injectivity of δ, the content of the bin, if any, 'to which x was assigned before execution of the block' is unaltered during that execution. In addition, a new variable is now associated with the mth bin after the above is done. This last requirement is the extra semantic requirement not occurring elsewhere. The variable-declaration is the only command construction in the definition of SSQ where

the righthand, 'δ', component of any term in the sequence gets changed from its previous value in the sequence. In other treatments, the computational sequence has only states s as terms, so δ never really changes. The interposing of bins in this definition is often avoided, but it does help at least one's intuition of what's going on.

(7) $\underline{C \ = \ \mathsf{begin\ proc}\ p(\vec{x} : \vec{v}) \equiv K\ \mathsf{corp}\ ;\ D_*\ ;\ C_*\ \mathsf{end}}$: Here the answer is

$$\prec \ s|\delta \ , \ SSQ(\mathsf{begin}\ D_*\ ;\ C_*\ \mathsf{end}/\pi', s|\delta) \ \succ$$

where

$$\pi'(q) \ := \ [\mathsf{if}\ q = p \ \ \mathsf{then}\ (K, (\vec{x} : \vec{v})) \ \ \mathsf{else}\ \pi(q)]) \ .$$

The intended effect here is that, when executing this block, any procedure call, to the procedure named p, will use K as that procedure, but the 'regime' given by π is restored immediately upon exit from the block.

This last four or five pages could all have been compressed into half a page. The wordiness above will hopefully help rather than irritate readers, especially non-CSers who take up the literature in the subject, which is occasionally closer to being machine-readable than human-readable! Always admirably brief, the choice seems to be between (readability and vagueness) or (unreadability and precision). By sacrificing brevity with a leisurely style, I hope to have largely avoided the other two sins.

Below in the proof system, we consider *'augmented' F-H statements*, which are strings of the form $F\{C/\pi\}G$, where π is, as above, a specification of which formal procedure is to be attached to each name from some finite set of procedure names (identifiers).

Here is the semantics of these F-H statements. We assume that the $1^{\underline{\text{st}}}$order (assertion) language has been fixed, so that **DTEN** and **CTEN** are meaningful, and also that an interpretation of the $1^{\underline{\text{st}}}$order language has been given, so that the semantic function SSQ can be defined.

Definition of truth of $F\{C/\pi\}H$:

In a fixed interpretation, say that $F\{C/\pi\}G$ is true if and only if, for any (s, δ) for which F is true at $s|\delta$, and for which $SSQ(C/\pi, s|\delta)$ is defined and is a finite sequence, we have that G is true at $OUT(C/\pi, s|\delta)$.

Note that an undefined $SSQ(C/\pi, s|\delta)$ is treated here exactly as if it were an infinite sequence, i.e. as if it produced an infinite loop.

Before listing the proof system's rules, here are some fundamental facts which one would intuitively expect to be true. In particular, part (b) below basically says that truth of a F-H statement is a property to do with states, s, (and not really to do with the functions, δ, which decide which variable should locate in which bin). Part (a) says that the semantics of a command really is a function of the map from variables to values in the interpretation, and doesn't essentially depend on the 'bins' as an intermediate stage in factoring that function. Much further along in separate places you will find parts (c), (d), (e) and (f). They have easy inductive proofs which we suppress, so they've been stated as close as possible to the location where they are needed (though it would be perfectly feasible to state and prove them here).

Theorem 8.6 (a) *Given* $C, \pi, \delta, s, \underline{\delta}, \underline{s}$ *, suppose that* $\underline{s} \circ \underline{\delta}(y) = s \circ \delta(y)$ *for all* $y \in FREE(C/\pi)$ *(in particular,* $\underline{\delta}$ *and* δ *are defined on all such* y*). Then, for all* i*, the term* $SSQ_i(C/\pi, s|\delta)$ *exists if and only if* $SSQ_i(C/\pi, \underline{s}|\underline{\delta})$ *exists and composing their two components, the two agree on all* $y \in FREE(C/\pi)$*.*

(b) *Given* $F\{C/\pi\}G$*, if there is a* $\underline{\delta}$ *whose domain includes all variables free in* F, G *and all of* $FREE(C/\pi)$*, and such that*

$$\forall \underline{s}\,,\ [F \text{ true at } \underline{s} \circ \underline{\delta} \text{ and } OUT(C/\pi, \underline{s}|\underline{\delta})] \text{ exists}] \implies G \text{ true at } OUT(C/\pi, \underline{s}|\underline{\delta}),$$

then, referring to δ*'s whose domain includes all free variables in* F, G *and all of* $FREE(C/\pi)$*, we have*

$$\forall \delta\, \forall s\,,\ [F \text{ true at } s \circ \delta \text{ and } OUT(C/\pi, s|\delta)] \text{ exists}] \implies G \text{ true at } OUT(C/\pi, s|\delta).$$

(This second display just says that $F\{C/\pi\}G$ is true in the interpretation.)

Proof. Part (a) is proved by induction on i, then for fixed i, by structural induction on C. The seven cases are quite straightforward from the definition of SSQ.

To deduce part (b) from (a), given $F\{C/\pi\}G$ and $\underline{\delta}$ as in the assumptions, let $s|\delta$ be a pair for which F is true at $s \circ \delta$ and $OUT(C/\pi, s|\delta)]$ exists. Now choose a state $\underline{s}$, requiring at least that $\underline{s}(\underline{\delta}(y)) = s(\delta(y))$ for all variables y free in F, G and in $FREE(C/\pi)$. Then part (a) gives us that the sequences

$$SSQ(C/\pi, \underline{s}|\underline{\delta}) \ = \ \prec \underline{s}_1|\underline{\delta}_1\ ,\ \underline{s}_2|\underline{\delta}_2\ ,\ \cdots\ ,\ \underline{s}_n|\underline{\delta}_n \succ$$

and

$$SSQ(C/\pi, s|\delta) \;=\; \prec s_1|\delta_1\ ,\ , s_2|\delta_2\ ,\ \cdots\ ,\ s_n|\delta_n \succ$$

have the same length, and, for all i, we have $\underline{s}_i \circ \underline{\delta}_i(y) = s_i \circ \delta_i(y)$ for all free variables y in F, G and C/π. But now, using this for $i = n$, which gives the OUT-state for the two sets of data, the truth of G at $\underline{s}_n \circ \underline{\delta}_n$ is the same as its truth at $s_n \circ \delta_n$, as required.

This concludes the specification of the semantics for **DTEN**, and so for the sublanguage **CTEN** singled out some paragraphs below. Because **DTEN** allows various possibilities which are forbidden to **CTEN**, it seems quite likely that the F-H proof system, given in the next few paragraphs, would be unsound as applied to the entire **DTEN**.

CTEN and Cook's complete F-H proof system for it.

Though perhaps useless, the system is meaningful for **DTEN**, so we'll continue to delay stating the caveats which delineate the sublanguage **CTEN**. The first four rules below match up exactly with the earlier ones for the language $\mathbf{ATEN}_{\mathcal{L}}$. As an exercise, the reader might give more exactly the connection; that is, show how to regard $\mathbf{ATEN}_{\mathcal{L}}$ plus its semantics as sitting inside **CTEN** plus *its* semantics.

Lines in a derivation using the system about to be defined will have the form $F\{C/\pi\}G$, (as well as some lines being $1^{\underline{\text{st}}}$order formulas). This at first might have seemed a bit fishy, since π had been introduced as a semantic concept. But just as F, C and G are names for strings of 'meaningless' symbols, so too, we shall take " / π " as short for a string

$$/\ p_1(\vec{x_1} : \vec{v_1}) \equiv C_1\ /\ p_2(\vec{x_1} : \vec{v_1}) \equiv C_2\ /\ \cdots\ /\ p_n(\vec{x_n} : \vec{v_n}) \equiv C_n\ \ ,$$

as explained earlier. So derivations can still be thought of as syntactically generated sequences of strings of meaningless symbols.

Validity of rules and soundness of a system of rules mean exactly the same as in the previous subsection.

The system of rules of inference :

(I) $$\frac{F\{C/\pi\}G \ , \ G\{\mathsf{begin}\ C_*\ \mathsf{end}\ /\ \pi\}H}{F\{\mathsf{begin}\ C; C_*\ \mathsf{end}\ /\ \pi\}H}$$

(II) $$\frac{\text{empty}}{F^{[x \mapsto t]}\{x \doteqdot t\ /\ \pi\}F}$$

(III) $$\frac{(F \wedge H)\{C/\pi\}F}{F\{\mathsf{whdo}(H)(C)/\pi\}(F \wedge \neg H)}$$

(IV) $$\frac{F \to F' \ , \ F'\{C/\pi\}G' \ , \ G' \to G}{F\{C/\pi\}G}$$

(V) $$\frac{\text{empty}}{F\{\mathsf{begin}\ \ \mathsf{end}\ /\ \pi\}F}$$

(VI) $$\frac{F^{[x \mapsto y]}\{\mathsf{begin}\ D_*; C_*\ \mathsf{end}\ /\ \pi\}G^{[x \mapsto y]}}{F\{\mathsf{begin\ new}\ x; D_*; C_*\ \mathsf{end}\ /\ \pi\}G}$$

if y is not free in F or G and is not in $FREE(\mathsf{begin}\ D_*; C_*\ \mathsf{end}\ /\ \pi)$.

(VII) $$\frac{F\{\mathsf{begin}\ D_*; C_*\ \mathsf{end}\ /\ \pi'\}G}{F\{\mathsf{begin}\ D; D_*; C_*\ \mathsf{end}\ /\ \pi\}G}$$

if $D = \mathsf{proc}\ p(\vec{x} : \vec{v}) \equiv K\ \mathsf{corp}$, and where π and π' agree except that $\pi'(p)$ is defined and equals $(K, (\vec{x} : \vec{v}))$, whatever the status of p with respect to π.

(VIII) $$\frac{F\{C/\pi'\}G}{F\{\mathsf{call}\ p(\vec{x} : \vec{v})\ /\ \pi\}G}$$

where the procedure declaration $\mathsf{proc}\ p(\vec{x} : \vec{v}) \equiv C\ \mathsf{corp}$ is included in π ; that is, $p \in \mathrm{dom}(\pi)$ and $\pi(p) = (C, (\vec{x} : \vec{v}))$, and where π' agrees with π, except that p is not in the domain of π'. CAUTION: The use of π' here would be inappropriate, except that we shall be disallowing recursive commands.

(IX) $$\frac{F\{C/\pi\}G}{F^{[\ \vec{y}\to\vec{r}\ ]}\{C/\pi\}G^{[\ \vec{y}\to\vec{r}\ ]}}$$

if no variable y_i nor any variable in any term r_i is in $FREE(C/\pi)$.
REMARK: This rule is more general than needed for the proof of completeness. The system would still be complete if we only had call-commands for C and only had $\vec{r}$ as a string of *variables*.

(X) $$\frac{F\{\mathsf{call}\ p(\vec{y}:\vec{w})/\pi\}G}{F^{[\ \vec{y}\to\vec{u}\ ;\ \vec{w}\to\vec{t}\ ]}\{\mathsf{call}\ p(\vec{u}:\vec{t})/\pi\}G^{[\ \vec{y}\to\vec{u}\ ;\ \vec{w}\to\vec{t}\ ]}}$$

if no u_i, is free in F or G, where $(\vec{y}:\vec{w})$ is a list of distinct variables (not necessarily related to the formal parameters $(\vec{x}:\vec{v})$ which π might give to p). Assume also that "$\vec{y}$ " $\cup$ "$\vec{w}$ " is disjoint from "$\vec{u}$ " $\cup$ "$\vec{t}$ ".

(XI) $$\frac{F\{C/\pi'\}G}{F\{C/\pi\}G}$$

if $\pi \subset \pi'$, i.e. they agree except that π might have a smaller domain.

Rule (XI) doesn't occur in [**C**], but neither does the " / π"-notation, so I suppose (XI) *couldn't* occur. But it does seem to be needed. The replacement for the " / π"-notation are phrases such as "with the understanding that all calls to p are according to D", and presumably the effects of rule (XI) are built into that phraseology.

A version of rule (XI) is given backwards (i.e. $\pi' \subset \pi$) in [**Cl1**], B2b), p.138.

Theorem 8.7 *Fix a* $1^{\underline{\text{st}}}$*order language* $\mathcal{L}$*. Let* $\mathbf{CTEN}_{\mathcal{L}}$ *be the sublanguage of* $\mathbf{DTEN}_{\mathcal{L}}$ *given by the list of restrictions* CV_* *just below this theorem. Let* I *be an interpretation of* $\mathcal{L}$*. Consider the F-H proof system obtained by combining* (I) *to* (XI) *with a decision oracle for for* $\mathcal{L}, I$*, i.e. use the often undecidable set of all* F *true in* I *as the set of premisses for all derivations. Then*

(i) that F-H system is sound; and

(ii) as long as $\mathcal{L}, I$ *is expressive with respect to* $\mathbf{CTEN}_{\mathcal{L}}$ *(which it certainly will be when* I *is finite, or when* $\mathcal{L}$ *is number theory and* $I = \mathbf{N}$*) that F-H proof system is also adequate (complete).*

Restrictions defining CTEN as a sublanguage of DTEN.

Firstly recall that already in **DTEN**, the variables $(\vec{x} : \vec{v})$ in a procedure declaration are all different from each other, as are the variables $\vec{u}$ in a procedure call command. And the latter cannot appear as variables in the terms $\vec{t}$.

Next we define $SCOM(K/\pi)$, the *set of subcommands*, to be $\{K\} \cup PSCOM(K/\pi)$, where the set, $PSCOM(K/\pi)$, of *proper* subcommands is defined by induction on the length of π and structural induction:

the empty set for assignment and 'do-nothing commands',

$$SCOM(C_1/\pi) \cup SCOM(\mathsf{begin}\ C_*\ \mathsf{end}/\pi) \quad \text{for} \quad K = \mathsf{begin}\ C_1; C_*\ \mathsf{end}$$

$$SCOM(\mathsf{begin}\ D_*; C_*\ \mathsf{end}/\pi) \quad \text{for} \quad K = \mathsf{begin}\ D_1; D_*; C_*\ \mathsf{end}$$

$$SCOM(C/\pi) \quad \text{for} \quad K = \mathsf{whdo}(H)(C)$$

and

$$SCOM(L^{[\vec{x}\to\vec{u}\ ;\ \vec{v}\to\vec{t}]}/\pi') \quad \text{for} \quad K = \mathsf{call}\ p(\vec{u} : \vec{t}) \ ,$$

where $\pi(p) = (L, (\vec{x} : \vec{v}))$ and π' agrees with π, except that p is removed from the domain. There are no proper subcommands if $p \notin$domain(π). (Note that the call command has no proper substrings which are subcommands, but may non-the-less have plenty of subcommands—see **8.6**(c) later where the need for this slightly more subtle and inclusive definition can be motivated. In fact, block commands can also have subcommands which are not, strictly speaking, substrings.)

As with free variables, to get the subcommands of a command on its own, just take π to be empty; that is, use $SCOM(C/\emptyset)$.

Definition of 'indirect activation sequence' , a.k.a. 'i.a.s.'.
An i.a.s. for C/π is a sequence :

$$\begin{array}{l}
\mathsf{occ}\ p_1(\vec{u}^{(1)} : \vec{t}^{(\ 1)})/\ /\ K_1(\vec{x}^{(1)} : \vec{v}^{(1)})\ /\ /\ \mathsf{occ}\ p_2(\vec{u}^{(2)} : \vec{t}^{(2)})/\ /\ K_2(\vec{x}^{(2)} : \vec{v}^{(2)}) \\
\qquad\qquad \cdots \\
\qquad\qquad \cdots \\
/\ /\ \mathsf{occ}\ p_\ell(\vec{u}^{(\ell)} : \vec{t}^{(\ell)})/\ /\ K_\ell(\vec{x}^{(\ell)} : \vec{v}^{(\ell)})\ /\ /\ \mathsf{occ}\ p_{\ell+1}(\vec{u}^{(\ell+1)} : \vec{t}^{(\ell+1)})/\ /\ \cdots \\
\qquad\qquad \cdots \\
\qquad\qquad \cdots \\
\mathsf{occ}\ p_n(\vec{u}^{(n)} : \vec{t}^{(n)})/\ /\ K_n(\vec{x}^{(n)} : \vec{v}^{(n)}) \ ,
\end{array}$$

(which may as well be finite)

such that the following hold.

(a) Its start term, $\mathsf{occ}\ p_1(\vec{u}^{(1)} : \vec{t}^{(1)})$, is an occurrence of $\mathsf{call}\ p_1(\vec{u}^{(1)} : \vec{t}^{(1)})$ as a subcommand in $SCOM(C/\emptyset)$.

(b) If $\mathsf{occ}\ p_1(\vec{u}^{(1)} : \vec{t}^{(1)})$ is an occurrence in a block of $SCOM(C/\emptyset)$ which includes a declaration of p_1, then that declaration is

$$\mathsf{proc}\ p_1(\vec{x}^{(1)} : \vec{v}^{(1)}) \equiv K_1\ \mathsf{corp}\ ;$$

otherwise $\pi(p_1) = K_1(\vec{x}^{(1)} : \vec{v}^{(1)})$.

(a+) For all $\ell \geq 1$, $\mathsf{occ}\ p_{\ell+1}(\vec{u}^{(\ell+1)} : \vec{t}^{(\ell+1)})$ is an occurrence of $\mathsf{call}\ p_{\ell+1}(\vec{u}^{(\ell+1)} : \vec{t}^{(\ell+1)})$ as a subcommand in $SCOM(K_\ell/\emptyset)$.

(b+) For all $\ell \geq 1$, if $\mathsf{occ}\ p_{\ell+1}(\vec{u}^{(\ell+1)} : \vec{t}^{(\ell+1)})$ is an occurrence in a block of $SCOM(K_\ell/\emptyset)$ which includes a declaration of $p_{\ell+1}$, then that declaration is

$$\mathsf{proc}\ p_{\ell+1}(\vec{x}^{(\ell+1)} : \vec{v}^{(\ell+1)}) \equiv K_{\ell+1}\ \mathsf{corp}\ ;$$

otherwise $\pi(p_{\ell+1}) = K_{\ell+1}(\vec{x}^{(\ell+1)} : \vec{v}^{(\ell+1)})$.

This is sometimes expressed informally by saying that calling the procedure p_1 *indirectly activates* all the p_ℓ for $\ell > 1$ (but perhaps activates other procedures as well, of course.)

Definition of CTEN/. Say that $C/\pi \in$ **CTEN/** if and only if the following six (really five) caveats hold :

$\underline{CV_1}$ A given procedure identifier p occurs in at most one procedure declaration within C together with all the indirect activation sequences for C/π.

REMARKS: It may occur as a declaration occurrence in a command which appears several times, once each in several i.a.s.'s for C/π . Also, a particular case of this caveat is that a given p is declared in C itself at most once, a sanitary requirement which simplifies the previous statement of the semantics, i.e. of SSQ. But also, in any i.a.s. for C/π, we have $p_n \neq p_1$. So this excludes 'recursive programming'.

In any indirect activation sequence for C/π, the caveats labelled with a "+" below hold :

$\underline{CV_2}$ If $\pi(p) = (K, (\vec{x} : \vec{v}))$ or if $\mathsf{proc}\ p(\vec{x} : \vec{v}) \equiv K\ \mathsf{corp}$ is a declaration in a subcommand in $SCOM(C/\emptyset)$, then the latter has no subcommands of the form $v_j \leftarrow t$ or $\mathsf{call}\ (\cdots v_j \cdots : \cdots)$; i.e. no v_j to the left of a colon.

$\underline{CV_2+}$ For $s \geq \ell \geq 1$, no subcommand in $SCOM(K_s/\emptyset)$, has the form $v_j^{(\ell)} \leftarrow t$, or the form $\mathsf{call}\ (\cdots v_j^{(\ell)} \cdots : \cdots)$.

$\underline{CV_3}$ If $\pi(p) = (K, (\vec{x} : \vec{v}))$ or if $\mathsf{proc}\ p(\vec{x} : \vec{v}) \equiv K\ \mathsf{corp}$ is a declaration in a subcommand in $SCOM(C/\emptyset)$, and if $\mathsf{call}\ (\vec{u} : \vec{t}) \in SCOM(C/\emptyset)$, then "$\vec{u}$ " $\cup$ "$\vec{t}$" is disjoint from $FREE(K/\pi)$.

$\underline{CV_3+}$ For $s \geq \ell \geq 1$, the set "$\vec{u}^{(\ell)}$ " $\cup$ "$\vec{t}^{(\ell)}$ " is disjoint from $FREE(K_s/\emptyset)$.

REMARK: The caveat CV_3 is just the case $s = 1 = \ell$ in CV_3+ (in one particular very short i.a.s.), but we state it separately because of the frequency to which it is referred later.

$\underline{CV_4+}$ For $s > \ell \geq 1$, the set "$\vec{x}^{(\ell)}$ " $\cup$ "$\vec{v}^{(\ell)}$ " is disjoint from $FREE(K_s/\emptyset)$.

REMARK: For $s > \ell = 1$, this says the formal parameters in the declaration of any called procedure are not free in any other procedure body which can be indirectly activated by the call.

REMARK: All the cases $\ell > 1$ above appear to be subsumed by simply saying that all K_s/π are in **CTEN/**; but that's a kind of circularity which is easy to hide when speaking glibly (non-technically)!

Proof of 8.7: Note that this proof occupies much of the remainder of this work. 'Logically', soundness should be proved first, but we'll start with adequacy, to make it clear why we need all those rules, before going to the trouble of checking their validity.

For adequacy, we proceed to show (in about 18 pages!) that if $F\{C/\pi\}G$ is true, then we can find a derivation for it. The proof is by induction on the cardinality of the domain of π, and then, for fixed such cardinality, structural induction on $C \in$ **CTEN**. (This takes the place of the need to discuss "substituting the body of p for each call to p, over-and-over till no calls remain", which depends on CV_1.) As with the definition of free variables, it's only when doing the hardest case, a call-command, that we need to appeal to the induction on the cardinality of the domain of π.

When C is an assignment command or whdo-command, or has the form $\mathsf{begin}\ C_1; C_*\ \mathsf{end}$, the arguments are virtually identical to the ones given in

the proof of **8.4**, modulo the somewhat different notation for semantics. So these cases will be omitted here, except for the following remarks. Note that this is where the "expressive" hypothesis in the theorem is needed, both for whdo-commands, and blocks without declarations (i.e. concatenation of commands). The required expressiveness is formulated and used as follows.

Given F and C/π, let

$$D_{F,C/\pi} := \{\, OUT(C/\pi, s|\delta) \;:\; F \text{ is true at } s|\delta \;\text{ and }\; SSQ(C/\pi, s|\delta) \text{ is finite} \,\}\,.$$

Then there is a formula $G = G_{F,C/\pi}$ such that G is true at $s' \circ \delta'$ if and only if $s'|\delta' \in D$. Thus, we get that $F\{C/\pi\}H$ is true iff $G \to H$ is true in the interpretation I.

To deal with begin $C_1; C_*$ end in this adequacy proof, use $C = C_1$ just above to produce an intermediate formula G, and proceed as in **8.4**.

REMARKS: It is interesting to analyze the counterexample given in [**C+**] with respect to where the proof in [**C**] goes wrong. In [**C+**], the two possible fixes are presented without any such analysis. This analysis here is a justification for inserting that last term in the sequence SSQ for variable-declaration-commands.

The example has the form $(C; D)$, where we simplify the concatenation notation with brackets instead of begin... end, and also drop all the $/\pi$, which are irrelevant to the point being made. Here, for a pair of distinct variables x and y,

$$C := \text{begin new } x \;;\; x \Leftarrow 1 \text{ end} \qquad \text{and} \qquad D := \text{begin new } x \;;\; y \Leftarrow x \text{ end} \;\;.$$

The semantics in [**C**] is the same as here with one exception: drop the last term in the defined SSQ here for variable-declaration-commands. In that semantics, δ never gets changed for the output, so SSQ is given simply as a sequence of s's. With that semantics (but not with ours, nor the fixes suggested in [**C+**]), it is easy to see that the F-H statement $Tr\{(C; D)\}y \approx 1$ is true. Here Tr is any logically valid formula. But that statement cannot be derived with the proof system. The completeness argument in [**C**] appears to break down for the case of concatenation, the difficulty turning on the following fine point concerning the existence of a strongest post-condition :

A re-wording of the fact we use above is

$$\forall F \forall C \forall \pi \exists G\; (\forall m \; G \text{ is true at } m \iff$$

$$\exists\, s|\delta \text{ with } F \text{ true at } s \circ \delta \text{ and } OUT(C/\pi, s|\delta) = s'|\delta' \text{ with } m = s' \circ \delta'.$$

In the semantics of [**C**] one can prove the corresponding

$$\forall F \forall C \forall \pi \forall \delta \exists G\; (\forall m \; G \text{ is true at } m \iff$$

$$\exists\, s \text{ with } F \text{ true at } s \circ \delta \text{ and } OUT(C/\pi, s|\delta) = s' \text{ with } m = s' \circ \delta.$$

But what is apparently needed for the completeness argument there is the more stringent requirement in which we just permute two quantifiers; that is, replace $\forall\delta\exists G$ with $\exists G\forall\delta$ just above. The example above shows by direct arguments that the more stringent requirement is false : On the one hand, the F-H statement $G\{D\}y \approx 1$ can only be true for a formula G which is never true (e.g. the negation of a logically valid formula). On the other hand, a formula G which satisfies the more stringent requirement, for C as in the example and $F = Tr$, is necessarily itself also logically valid. But, as in the proof of **8.4**, such a strongest postcondition G must make $G\{D\}H$ true, given that $Tr\{(C;D)\}H$ is true. So no strongest postcondition in the sense of the more stringent requirement can exist, for $F = Tr$ and C in the example.

Using our semantics here with H being $y \approx 1$, the statement $Tr\{(C;D)\}H$ simply isn't true, so the contradiction doesn't arise.

When $C =$ begin end, axiom (V) gives a derivation of $F\{C/\pi\}F$. The truth of $F\{C/\pi\}G$ shows that the formula $F \to G$ is true in I. Now just apply rule (IV) with numerator

$$F \to F \quad , \quad F\{C/\pi\}F \quad , \quad F \to G$$

to get the required derivation.

The next two of the final three cases are straightforward, but a little intricate in details, so we'll give them both carefully. We still won't need any of the caveats CV carving **CTEN** out as a subset of **DTEN** for these two. And the argument in each these cases rests on the one obviously relevant rule.

Suppose that $C =$ begin new $x; D_*; C_*$ end . Define $C_- :=$ begin $D_*; C_*$ end. Assume that $F\{C/\pi\}G$ is true. Fix any variable y different from x and not occurring in any of F or G or D_* or C_*. We shall prove that

$$(*) \qquad F^{[x\to y]}\{C_-/\pi\}G^{[x\to y]} \quad \text{is true.}$$

Then, by structural induction, the displayed F-H statement has a derivation. So the proof is completed by an application of rule (VI) to get the required derivation of $F\{C/\pi\}G$.

Let $\mathcal{F}$ be the set of variables which occur in F and/or G and/or $FREE(C/\pi)$. Choose δ' , with domain equal to $\mathcal{F} \cup \{x, y\}$, by first defining it arbitrarily (but injectively!) on $\mathcal{F}\setminus\{x, y\}$, then defining $\delta'(y) = \text{bin}_a$ and $\delta'(x) = \text{bin}_{a+1}$, for some a with $a > i$ for all i with $\text{bin}_i \in \underline{\delta}(\mathcal{F} \setminus \{x, y\})$.

To prove $(*)$, by **8.6**(b), it suffices to prove the following: given s such that $F^{[x \mapsto y]}$ is true at $s \circ \delta'$, and that $SSQ(C_-/\pi, s|\delta')$ is a finite sequence, we must show

$(**)$ $\qquad G^{[x \mapsto y]}$ is true at $OUT(C_-/\pi, s|\delta')$.

Define δ so that it is defined exactly on all variables other than y in the domain of δ', with $\delta(x) = \delta'(y) = \mathrm{bin}_a$, and δ agrees with δ' everywhere else on its domain.

Now δ' (except for being defined on y, which is irrelevant, as y doesn't occur in C) is obtained from δ as in the definition of SSQ for C in terms of SSQ for C_-. Thus $LOUT(C/\pi, s|\delta) = LOUT(C_-/\pi, s|\delta')$.

Since $s \circ \delta$ and $s \circ \delta'$ agree on all variables except that $s \circ \delta(x) = s \circ \delta'(y)$ (and except that $s \circ \underline{\delta}'(x)$ is defined but irrelevant), and since $F^{[x \mapsto y]}$ is true at $s \circ \delta'$, it follows that F is true at $s \circ \delta$.

Since $F\{C/\pi\}G$ is true, it now follows that G is true at $OUT(C/\pi, s|\delta)$.

Using this, we now obtain that $G^{[x \mapsto y]}$ is true at $OUT(C_-/\pi, s|\delta')$, which is $(**)$, as required. This last step is immediate because the corresponding $LOUT$s agree (noted above), and because the $ROUT$s agree except that

$$ROUT(C/\pi, s|\delta)(x) \;=\; \delta''(x) \;=\; \delta(x) \;=\; \delta'(y) \;=\; ROUT(C_-/\pi, s|\delta')(y)\ ,$$

and they may disagree on variables beyond those occurring free in G. (Recall that $ROUT(C/\pi, s|\delta)$ is the δ'' in the SSQ definition for variable-declaration-commands.)

For the penultimate case, suppose that $C = \mathsf{begin}\ D; D_*; C_*\ \mathsf{end}$ for some procedure declaration D. Define $C' = \mathsf{begin}\ D_*; C_*\ \mathsf{end}$. As usual, we are given that $F\{C/\pi\}G$ is true, and we want a derivation for it.

Define π' exactly the way π' and π are related in rule (VII); that is, they agree except that $\pi'(p)$ is defined and equals $(K, (\vec{x} : \vec{v}))$, whatever the status of p with respect to π, where D is $\mathsf{proc}\ p(\vec{x} : \vec{v}) \equiv K\ \mathsf{corp}$. By the definition of $SSQ(C/\pi, s|\delta)$ in terms of $SSQ(C'/\pi', s|\delta)$, it follows that

$$OUT(C/\pi, s|\delta) = OUT(C'/\pi', s|\delta)\ .$$

(Again, the two SSQ sequences agree, except for a spurious copy of $s|\delta$ at the beginning of one of them.)

From the display and the truth of $F\{C/\pi\}G$, it is immediate that $F\{C'/\pi'\}G$ is true. And so, by induction, it has a derivation. Now application of rule (VII) gives a derivation for $F\{C/\pi\}G$, as required.

Finally we come to the delicate case where C is a call-command. Here we apparently need the full force of the caveats which disallow many **DTEN**-commands. As far as I can determine, cases of languages closer to full ALGOL than **CTEN**, and proof systems which are sound and complete for the F-H statements corresponding to such languages, and where correct and complete mathematical proofs have been written out for those facts, are relatively thin on the ground, and **[Old1]** seems the best choice for getting at least some details. This is very technical work, and ordinary mortals cannot become experts overnight! We refer the reader again to the quotes in the last subsection of this paper. The proof below follows that in **[C]**, filling in details, and correcting a couple of errors, as noted earlier.

For the case of the structural inductive adequacy proof where C is a call-command, and also for later purposes, we need the following lemmas.

Lemma 8.8 *Let C/π be any command in* **DTEN/**. *Assume that*

$$\delta_1|_{FREE(C/\pi)} \;=\; \delta_2|_{FREE(C/\pi)} \qquad \text{where } FREE(C/\pi) \subset \mathrm{dom}(\delta_1) \cap \mathrm{dom}(\delta_2)\ ,$$

and that the minimum m greater than all i for which bin_i is in the image of δ_1 is the same as that for δ_2. Then

$$LOUT(C/\pi, s|\delta_1) \;=\; LOUT(C/\pi, s|\delta_2)\ .$$

(Recall that $LOUT$ is the left-half, the 's-half', of OUT.)

Proof. This is an induction 'on SSQ' (very similar to the one in the proof of **8.6**(a)) showing that the left-halves of SSQ_n agree for all n. The case $C = \mathsf{begin\ new}\ x; D_*; C_*\ \mathsf{end}$ is the only one where the 'answer' uses a different δ than the starter. In that case, the conditions above, relating the starters δ_1 and δ_2, insure that the same conditions relate the two δ's used in calculating SSQ inductively. So the induction goes through. (This lemma is used only once below, non-essentially, so we won't flesh this proof out.)

Definition. We say that "$\underline{s}|\underline{\delta}$ is $(\vec{x}\mapsto \vec{u}\ ,\ \vec{v}\mapsto \vec{t})$—matched to $s|\delta$", if and only if the following four conditions hold:

(A) $[y \in \mathrm{dom}(\underline{\delta})\cap\mathrm{dom}(\delta)$ and $y \notin$ "$\vec{x}$"$\cup$"$\vec{v}$"$\cup$"$\vec{u}$"$]$ $\implies$ $\underline{s}\circ\underline{\delta}(y) = s\circ\delta(y)$;

(B) $$\forall k\ ,\quad \underline{s}\circ\underline{\delta}(x_k)\ =\ s\circ\delta(u_k)\ ;$$

(C) $$\forall j\ ,\quad \underline{s}\circ\underline{\delta}(v_j)\ =\ t_j^{s\circ\delta}\ ;$$

(D) $\underline{s}(\mathrm{bin}_{\underline{i}}) = s(\mathrm{bin}_i)$ for $\underline{i} - \underline{m} = i - m \geq 0$ where $\underline{m}$ and m are the smallest subscripts of bins larger than the subscripts of all bins in the images of $\underline{\delta}$, δ respectively.

NOTE: Including $\cdots\cup$ "$\vec{u}$ " here in (A) disagrees with **[C]**, but his analogue of **8.10** below is false.

Lemma 8.9 *Assume given a formula F [and/or a term e], as well as $\underline{s}, \underline{\delta}, s, \delta, \vec{x}, \vec{u}, \vec{v}, \vec{t}$ such that :*

(i) $\underline{s}|\underline{\delta}$ is $(\vec{x}\mapsto \vec{u}\ ,\ \vec{v}\mapsto \vec{t})$—matched to $s|\delta$;

(ii) $\underline{\delta}$ and δ are defined for all variables free in the formulas F and $F^{[\vec{x}\mapsto\vec{u}\ ,\ \vec{v}\mapsto\vec{t}]}$ [and/or all variables in the terms e and $e^{[\vec{x}\mapsto\vec{u}\ ,\ \vec{v}\mapsto\vec{t}]}$] ;

(iii) the variables in "$\vec{u}$" do not appear in F and/or e.

Then

$$F^{[\vec{x}\mapsto\vec{u}\ ;\ \vec{v}\mapsto\vec{t}]} \text{ is true at } s\circ\delta \quad\Longleftrightarrow\quad F \text{ is true at } \underline{s}\circ\underline{\delta}\ ;$$

[and/or

$$(e^{[\vec{x}\mapsto\vec{u}\ ;\ \vec{v}\mapsto\vec{t}]})^{s\circ\delta}\ =\ e^{\underline{s}\circ\underline{\delta}}\ ]\ .$$

NOTE: Requiring *(iii)* here (because of the $\cdots\cup$"$\vec{u}$ " in (A) of the definition) weakens this compared to the corresponding Lemma 2 in **[C]**. But all five applications of this lemma below do in fact work alright. And Lemma 3 in **[C]** turns out to actually be false because $\cdots\cup$ "$\vec{u}$ " wasn't required in **[C]**—see the example at the end of the six page proof of **8.10** just ahead.

Proof. This is basic 1$^{\underline{\mathrm{st}}}$order logic. It uses (A), (B) and (C) (but not (D) of course). The point is simply that substitutions into formulas and terms give answers whose semantics, relative to that prior to substitution, vary only with respect to the variables involved in that substitution.

Lemma 8.10 *Let* $[K, \pi, (\vec{x} : \vec{v}), (\vec{u} : \vec{t})]$ *be such that :*
(i) the usual disjointness of the variables in $(\vec{x} : \vec{v})$ *and in* $(\vec{u} : \vec{t})$ *holds ;*
(ii) $K/\pi \in$ **CTEN/** *;*
(iii) for any $p \in PRIDE$*, every i.a.s. for* call $p(\vec{u} : \vec{t})/\pi$ *which begins as* occ $p(\vec{u} : \vec{t})//K(\vec{x} : \vec{v})//\cdots$ *satisfies the caveats* CV_2+, CV_3+ *and* CV_4+ *in the definition of* **CTEN**.

[This is more-or-less saying that K is a procedure which can be "legally" called within **CTEN/**. The specification of the second term in the i.a.s. amounts to requiring that $\pi(p) = (K, (\vec{x} : \vec{v}))$.]
Assume that $\underline{s}|\underline{\delta}$ *is* $(\vec{x} \to \vec{u}\ ,\ \vec{v} \to \vec{t})$—matched to $s|\delta$. *Then*

$$OUT(K/\pi\ ,\ \underline{s}|\underline{\delta}) \text{ is } (\vec{x} \to \vec{u}\ ,\ \vec{v} \to \vec{t}) - \text{matched to } \quad OUT(K^{[\vec{x} \to \vec{u}\ ;\ \vec{v} \to \vec{t}]}/\pi\ ,\ s|\delta).$$

Its three applications. It is readily checked that *(i),(ii)* and particularly *(iii)* hold for $[K, \pi, (\vec{x} : \vec{v}), (\vec{u} : \vec{t})]$ in the following three examples, (α), (β) and (γ). These are the three situations [**8.11**, validity of (IX),(X)] where the lemma will be applied below. Checking these three may also help to familiarize the reader with the technicalities. In **[C]**, the entire set of assumptions is stated "where K is any statement such that $p(\vec{x} : \vec{v})$ proc K could be a legal declaration for a legal statement call $p(\vec{u} : \vec{t})$". But I could not find a less technical way to formulate this so that both its proof and its applications inspired reasonable confidence.

(α) For some p, call $p(\vec{u} : \vec{t})/\pi \in$**CTEN/** and $\pi(p) = (K, (\vec{x} : \vec{v}))$.

(β) $[C, \pi, (\emptyset : \vec{y}), (\emptyset : \vec{r})]$ where $C/\pi \in$**CTEN/** and "$\vec{y}$ " $\cup$ "$\vec{r}$ " is disjoint from $FREE(C/\pi)$.

(γ) $[L, \pi, (\vec{y} : \vec{w}), (\vec{u} : \vec{t})]$ where $(\vec{y} : \vec{w}) \in IDE^{*!}$, "$\vec{y}$ " $\cup$ "$\vec{w}$ " is disjoint from "$\vec{u}$ " $\cup$ "$\vec{t}$ ", and $L = K^{[\ \vec{x} \to \vec{y}\ ;\ \vec{v} \to \vec{w}\ ]}$ with $[K, \pi, (\vec{x} : \vec{v}), (\vec{u} : \vec{t})]$ as in (A).

Notation. Very occasionally, we shall need to separate $OUT(K/\pi, s|\delta)$ into its two halves, which we'll then specify by putting an L for "left" and an R for "right" in front of the OUT. That is,

$$OUT(K/\pi, s|\delta) \ =: \ LOUT(K/\pi, s|\delta) \mid ROUT(K/\pi, s|\delta)\ .$$

Proof of 8.10. This is an induction on the cardinality of $SCOM(K/\pi)$, and then on K, which has several non-obvious cases (of the usual seven cases), including

the cases of variable-declaration commands and assignment commands, where the caveats CV singling out **CTEN** from **DTEN** play a crucial role, and worse, if anything, the case of a call-command, where the $CV+$ are crucial. We'll leave the case of assignment commands to the last, for reasons indicated in the small print just after the proof. Unfortunately, this is where we (unavoidably it seems) reach an orgasm of technicalities. So some readers may prefer to skip the 9-page proof for now. On the other hand, this is really where the reasons for the restriction from **DTEN** to **CTEN** become manifest, at least if one cannot think of alternative lines of argument (or *alternate* lines of argument, for USers).

To help express the various cases of the proof readably, let us denote the displayed OUTputs in the lemma as

$$OUT(K/\pi\ ,\ \underline{s}|\underline{\delta}) = \underline{\sigma}_K(\underline{s}|\underline{\delta})$$

and

$$OUT(K^{[\vec{x}\to\vec{u}\ ;\ \vec{v}\to\vec{t}]}/\pi\ ,\ s|\delta) = \sigma_K(s|\delta)\ .$$

And let's suppress the $\vec{x}\to\ \vec{u}\ ;\ \vec{v}\to\ \vec{t}$ from the notation. And finally let's shorten "—matched" to just "–m–"

So we must prove

$$\underline{s}|\underline{\delta}\ \text{–m–}\ s|\delta \quad\Longrightarrow\quad \underline{\sigma}_K(\underline{s}|\underline{\delta})\text{–m–}\sigma_K(s|\delta)\ .$$

Note before starting that condition (iii) on K in the lemma is inherited by any subcommand, so it is only the penultimate of the seven cases below (the call-command) where we need to check that the inductive assumption actually applies to the relevant command.

Case when $K = \mathsf{begin\ end.}$

This is beyond the pale of triviality.

Case when $K = \mathsf{begin}\ C_1; C_*\ \mathsf{end}.$

Applying the structural induction twice, $\underline{s}|\underline{\delta}\ \text{–m–}\ s|\delta$ implies that $\underline{\sigma}_{C_1}(\underline{s}|\underline{\delta})\text{–m–}\sigma_{C_1}(s|\delta)$. But then the latter implies that

$$\underline{\sigma}_{\mathsf{begin}\ C_*\ \mathsf{end}}(\underline{\sigma}_{C_1}(\underline{s}|\underline{\delta}))\ \text{–m–}\ \sigma_{\mathsf{begin}\ C_*\ \mathsf{end}}(\sigma_{C_1}(s|\delta)).$$

However, the definition of SSQ for K in terms of C_1 and C_* immediately shows that the last statement is identical with the required one, namely

$$\underline{\sigma}_K(\underline{s}|\underline{\delta})\text{–m–}\sigma_K(s|\delta)\ .$$

Case when $K = \mathsf{whdo}\ (H)(C)$.

Modulo one crucial observation, this is really all the instances of the previous case of the form $\mathsf{begin}\ C; C; \cdots; C\ \mathsf{end}$. But we also must observe that, in the relevant cases (where the whdo-command terminates), the formulas H and $H^{[\vec{x}\to\vec{u}\ ;\ \vec{v}\to\vec{t}]}$ become false after exactly the same number of iterations in both cases, by **8.9** and induction. This uses **8.9** applied to the formula H, and our extra condition *(iii)* there holds, because no u_i can occur free in K, by CV_3+, $\ell = s = 1$, so in particular, in H. (The double-underlined is first in our recorded list of the assumptions about the given data which are needed to push the proof through.)

Case when $K = \mathsf{begin}\ D; D_*; C_*\ \mathsf{end}$, with D a procedure declaration.

Let $K' = \mathsf{begin}\ D_*; C_*\ \mathsf{end}$. Recall, from the definition of SSQ for this case, that we modified π to π' to express the state sequence for K in terms of that for K'. In fact, π' agrees with π, except that it maps the procedure name in D to its body and formal parameters, which need no notation here. We shall use π and π' in the subscripts on the σ's in the obvious way. Then the definition of SSQ just referred to gives

$$\underline{\sigma}_{K/\pi}(\underline{s}|\underline{\delta}) = \underline{\sigma}_{K'/\pi'}(\underline{s}|\underline{\delta}) \quad \text{and} \quad \sigma_{K/\pi}(s|\delta) = \sigma_{K'/\pi'}(s|\delta) \quad .$$

But now the inductive hypothesis applied to K' gives exactly the required statement about K.

Case when $K = \mathsf{begin\ new}\ x; D_*; C_*\ \mathsf{end}$.

Let $K' = \mathsf{begin}\ D_*; C_*\ \mathsf{end}$. As in the definition of SSQ for K in terms of K', define $\underline{\delta}', \delta'$ to agree with $\underline{\delta}, \delta$, except that they map the variable x to $\mathrm{bin}_{\underline{m}}$, bin_m, respectively. And define $\underline{\delta}'', \delta''$ to agree with $\underline{\delta}, \delta$, except that they map variables $\underline{z}, z$ to $\mathrm{bin}_{\underline{m}}$, bin_m, respectively, where $\underline{z}, z$ are the least variable after the domains of $\underline{\delta}, \delta$ respectively.

If x occurs in $(\vec{x} : \vec{v})$, consider the simultaneous substitution in which the one for x is omitted, but which is otherwise identical to $\vec{x}\to\ \vec{u}\ ;\ \vec{v}\to\ \vec{t}$. Use notation "–m*–" for the matching corresponding to that (possibly restricted) substitution. Direct from the definition of freeness, x is not free in K, so these two (possibly different) substitutions have the same effect on K

Now, direct from the definition of matching, that $\underline{s}|\underline{\delta}\ -\mathrm{m}-s|\delta$ gives that

$$\underline{s}|\underline{\delta}'\ -\mathrm{m}-s|\delta' \ .$$

The equations for the latter are identical with the equations for the former, except for the following : the equation $\underline{s}(\mathrm{bin}_{\underline{m}}) = s(\mathrm{bin}_m)$ from part (D) of the former isn't part of (D) for the latter, but is used rather for one new equation needed to verify part (A) or (B) or (C) of the latter, depending respectively on whether the variable x is not in "$\vec{x}$ " $\cup$ "$\vec{v}$ " $\cup$ "$\vec{u}$ ", or is in "$\vec{x}$ ", or is in "$\vec{v}$ ". (If it's in "$\vec{u}$ ", there is no extra equation to worry about.)

Then, applying the inductive hypothesis to K' and using the display immediately above,

$$\underline{\sigma}_{K'}(\underline{s}|\underline{\delta}') \text{ –m}^*\text{– } \sigma_{K'}(s|\delta').$$

Now the definition of SSQ for K in terms of K' shows the displayed claim just above to be the same as

$$(*) \qquad LOUT(K/\pi, \underline{s}|\underline{\delta}) \mid \underline{\delta}^* \text{ –m}^*\text{– } LOUT(K^{[\vec{x}\to\vec{u}\ ;\ \vec{v}\to\vec{t}]}, s|\delta) \mid \delta^*,$$

for $\underline{\delta}^*, \delta^*$ defined by the right-hand sides of the first and third displays just below.

The claim about $(*)$ holds because

$$\underline{\sigma}_{K'}(\underline{s}|\underline{\delta}') = OUT(K'/\pi, \underline{s}|\underline{\delta}') \ = \ (\text{say})\ LOUT(K/\pi, \underline{s}|\underline{\delta})|\underline{\delta}^* \ ,$$

where

$$LOUT(K/\pi, \underline{s}|\underline{\delta})|\underline{\delta}'' = OUT(K/\pi, \underline{s}|\underline{\delta}) = \underline{\sigma}_K(\underline{s}|\underline{\delta}) \ ;$$

and because

$$\sigma_{K'}(s|\delta') = OUT(K'^{[\vec{x}\to\vec{u}\ ;\ \vec{v}\to\vec{t}]}/\pi, s|\delta') \ = \ (\text{say})\ LOUT(K^{[\vec{x}\to\vec{u}\ ;\ \vec{v}\to\vec{t}]}/\pi, s|\delta) \mid \delta^* \ ,$$

where

$$LOUT(K^{[\vec{x}\to\vec{u}\ ;\ \vec{v}\to\vec{t}]}/\pi, s|\delta)|\delta'' = OUT(K^{[\vec{x}\to\vec{u}\ ;\ \vec{v}\to\vec{t}]}/\pi, s|\delta) = \sigma_K(s|\delta) \ .$$

The middle equality in the last display depends on the fact that

$$K'^{[\vec{x}\to\vec{u}\ ;\ \vec{v}\to\vec{t}]} = (K^{[\vec{x}\to\vec{u}\ ;\ \vec{v}\to\vec{t}]})' \ .$$

By the second and fourth displays above, proving the required

$$(**) \qquad \underline{\sigma}_K(\underline{s}|\underline{\delta}) \text{ –m– } \sigma_K(s|\delta)$$

amounts to changing, in $(*)$, the $\underline{\delta}^*, \delta^*$ in $(**)$ to $\underline{\delta}'', \delta''$, and getting rid of the "*" on the matching. This where we get mildly painstaking. The crucial

observation in each case is that the execution of K using $\underline{\delta}$, because it starts with declaring "new x", has no effect on the contents of the bin $\underline{\delta}(x)$; that is, all the states in its SSQ-sequence map $\underline{\delta}(x)$ to the same value.

Here is a formal version of the general result needed for this observation, which is intuitively obvious, and can be proved very easily for the state SSQ_n by induction on n, then by structural induction :

8.6(c) *For any $(C/\pi, s|\delta)$ whose computation sequence exists, if*

$$SSQ(C/\pi, s|\delta) =\prec s_1|\delta_1 \ , \ s_2|\delta_2 \ , \ \cdots \succ \ ,$$

then, for all but finitely many bins b, we have $s_n(b) = s(b)$ for all n. In particular, this can fail to hold only for $b = \delta(z)$ for variables z for which some assignment command $z \leftarrow\!\!\div \ e$ is in $SCOM(C/\pi)$. In particular, this unaffectedness of $s \circ \delta(z)$ by execution of C/π holds for all variables z not in $FREE(C/\pi)$

$$[\text{ since } \quad z \leftarrow\!\!\div \ e \ \in SCOM(C/\pi) \implies z \in FREE(C/\pi) \ ] \ .$$

Recall that $SCOM(K/\pi)$ is the set of subcommands, defined earlier, and it includes subcommands in all procedures that might get called and executed. In each case below, we are using $\underline{\sigma}$ [and σ respectively] as short for the left half of $\underline{\sigma}_K(\underline{s}|\underline{\delta})$ [and $\sigma_K(s|\delta)$ resp.]

Case (a) : $x \notin$ "$\vec{x}$ " $\cup$ "$\vec{v}$ " $\cup$ "$\vec{u}$ ".

Here the two substitutions are actually the same. With one switch, most equations for $(**)$ are the same as those for $(*)$. For (D) in $(**)$, we need the bottom case

$$\sigma(\text{bin}_m) = \sigma \circ \delta'(x) = \underline{\sigma} \circ \underline{\delta}'(x) = \underline{\sigma}(\text{bin}_{\underline{m}}) \ ,$$

the middle equality being part of (A) for $(*)$. Also there is the additional equation in (A) for $(**)$, namely

$$\underline{\sigma} \circ \underline{\delta}(x) = \underline{s} \circ \underline{\delta}(x) = s \circ \delta(x) = \sigma \circ \delta(x) \ .$$

The two outside equalities are the principle (concerning the computation not affecting most bins) discussed just above; that is, use **8.6**(c). The middle one is the basic matching hypothesis in this lemma.

Case (b) : $x \in$ "$\vec{x}$ ".

With two exceptions (the displays below), the equations for $(**)$ are the same as those for $(*)$. For (D) we need the bottom case

$$\sigma(\mathrm{bin}_m) = \sigma \circ \delta'(x) = \underline{\sigma} \circ \underline{\delta}'(x) = \underline{\sigma}(\mathrm{bin}_{\underline{m}}) \quad ,$$

the middle equality being part of (A) for $(*)$. And, for (B), if $x_k = x$, we get

$$\underline{\sigma} \circ \underline{\delta}(x_k) = \sigma \circ \delta(u_k)$$

because the same holds with $s, \underline{s}$ in place of $\sigma, \underline{\sigma}$, and because the bins $\underline{\delta}(x_k)$ and $\delta(u_k)$ are unaffected by the execution of K and $K^{[\vec{x}\to\vec{u}\ ;\ \vec{v}\to\vec{t}]}$ respectively. Note that u_k is not free in the latter command because it's not free in K, it doesn't get substituted for x because x is not free in K, and it's different from all the other u_i's and the variables in the t_j's (even if we were in **DTEN**).

Case (c) : $x \in$ "$\vec{v}$ ".

With two exceptions (the displays below), the equations for $(**)$ are the same as those for $(*)$. For (D) we need the bottom case

$$\sigma(\mathrm{bin}_m) = \sigma \circ \delta'(x) = \underline{\sigma} \circ \underline{\delta}'(x) = \underline{\sigma}(\mathrm{bin}_{\underline{m}}) \quad ,$$

the middle equality being part of (A) for $(*)$. And, for (C), if $v_j = x$, we get

$$\underline{\sigma} \circ \underline{\delta}(v_j) = t_j^{\sigma\circ\delta} \ ,$$

as required, because the same holds with $s, \underline{s}$ in place of $\sigma, \underline{\sigma}$, and because the bins $\delta(z)$ for all variables z in t_j (and $\underline{\delta}(v_j)$ respectively) are unaffected by the execution of $K^{[\vec{x}\to\vec{u}\ ;\ \vec{v}\to\vec{t}]}$ (and K respectively). The argument for this last claim is as follows : These variables z *can* be free in $K^{[\vec{x}\to\vec{u}\ ;\ \vec{v}\to\vec{t}]}$, but, because v_j is not free in K, so t_j doesn't get substituted for it, this can only happen with another v_i being replaced by some t_i, for $i \neq j$, where t_i and t_j have the variable z in common. But then no $z \mathrel{\div\!\!\leftarrow} e$ is in $SCOM(K^{[\vec{x}\to\vec{u}\ ;\ \vec{v}\to\vec{t}]})$ because v_i cannot occur to the left of a " $\mathrel{\div\!\!\leftarrow}$ " in K, by CV_2, and so $\delta(z)$ is unaffected by executing $K^{[\vec{x}\to\vec{u}\ ;\ \vec{v}\to\vec{t}]}$, as required.

Case (d) : $x \in$ "$\vec{u}$ ".

With one exception (the same old one!), the equations for $(**)$ are the same as those for $(*)$. For (D) we need the bottom case

$$\sigma(\mathrm{bin}_m) = \sigma \circ \delta'(x) = \underline{\sigma} \circ \underline{\delta}'(x) = \underline{\sigma}(\mathrm{bin}_{\underline{m}}) \quad ,$$

the middle equality being part of (A) for $(*)$.

This completes the case of a command which is a 'variable-declaration block'.

Case when $K = \mathsf{call}\ p'(\vec{u}' : \vec{t}')$.

If p' is not in the domain of π, the displayed OUT-states in the theorem do not exist and there's nothing to prove.

Otherwise, let $\pi(p') = (L, (\vec{x}' : \vec{v}'))$. Now, assuming that the inductive hypothesis applies to $L_1 := L^{[\vec{x}' \to \vec{u}' \ , \ \vec{v}' \to \vec{t}']}$, the proof is completed in this case as follows. We have, for some N, using the definition of SSQ for calls,

$$\underline{\sigma}_K(\underline{s}|\underline{\delta}) \ := \ OUT(K/\pi, \underline{s}|\underline{\delta}) \ = \ SSQ_N(K/\pi, \underline{s}|\underline{\delta}) \ =$$

$$SSQ_{N-1}(L_1/\pi, \underline{s}|\underline{\delta}) \ =: \ \underline{\sigma}_{L_1}(\underline{s}|\underline{\delta}) \ .$$

Define

$$u''_j := u'^{[\vec{x} \to \vec{u} \ , \ \vec{v} \to \vec{t}]}_j \quad \text{and} \quad t''_i := t'^{[\vec{x} \to \vec{u} \ , \ \vec{v} \to \vec{t}]}_i \quad .$$

Because "$\vec{x}$ " $\cup$ "$\vec{v}$ " is disjoint from $FREE(L/\pi)$ by CV_4+ , a direct elementary argument shows that

$$L^{[\vec{x}' \to \vec{u}'' \ , \ \vec{v}' \to \vec{t}'']} \ = \ L_1^{[\vec{x} \to \vec{u} \ , \ \vec{v} \to \vec{t}]} \ .$$

Thus

$$\sigma_K(s|\delta) \ := \ OUT(K^{[\vec{x} \to \vec{u} \ , \ \vec{v} \to \vec{t}]}/\pi, s|\delta) \ = \ OUT(\mathsf{call}\ p'(\vec{u}'' : \vec{t}'')\ /\pi, s|\delta) \ =$$

$$OUT(L^{[\vec{x}' \to \vec{u}'' \ , \ \vec{v}' \to \vec{t}'']}\ /\pi, s|\delta) \ = \ OUT(L_1^{[\vec{x} \to \vec{u} \ , \ \vec{v} \to \vec{t}]}\ /\pi, s|\delta) \ =: \ \sigma_{L_1}(s|\delta) \ .$$

So we have reduced the matching question from the required K to the command L_1.

The set $SCOM(L_1/\pi)$ is a proper subset of $SCOM(K/\pi)$, with cardinality one less. So the inductive hypothesis does indeed apply to L_1, completing the proof here, once we have argued that $[K, \pi, (\vec{x} : \vec{v}), (\vec{u} : \vec{t})]$ satisfying

(iii) implies that $[L_1, \pi, (\vec{x} : \vec{v}), (\vec{u} : \vec{t})]$ also satisfies *(iii)* (in the statement of **8.10**).

To do that, consider an i.a.s. for call $p(\vec{u} : \vec{t})/\pi$ as follows:

$$\mathsf{occ}\ p(\vec{u} : \vec{t})/\ /\ L_1(\vec{x} : \vec{v})/\ /\ \mathsf{occ}\ p_2(\vec{u}^{(2)} : \vec{t}^{(2)})/\ /\ \cdots$$

The third term is an occurrence of a call in $L_1 = L^{[\vec{x}' \to \vec{u}'\ ,\ \vec{v}' \to \vec{t}']}$. The corresponding call occurrence in L cannot involve the variables in "$\vec{x}'$ " $\cup$ "$\vec{v}'$ " because of CV_4+ for K, applied to i.a.s.'s which have K in their 2nd term and L in their 4th term. Thus the third term in the display is identical with its corresponding call occurrence in L. So all the conditions with $s > 1$ automatically hold in the case of L_1 because they hold for the above i.a.s.'s which have K in their 2nd term and L in their 4th term.

Thus what remains to check are the conditions on $[L_1, \pi, (\vec{x} : \vec{v}), (\vec{u} : \vec{t})]$ which are CV_2+ and CV_3+ for $s = \ell = 1$. For the latter, the set "$\vec{u}$ " $\cup$ "$\vec{t}$ " is disjoint from $FREE(L_1/\emptyset)$ because the same holds for L. For the former, no v_i occurs left of a colon in L_1 again because the same holds for L.

This completes the case of a call-command.

Case when K is the assignment command $z \div\!\!= e$.

Here there is no induction to rest on, and one must simply go back to the definition of matching and check the equations carefully.

Firstly, our assignment command, $z \div\!\!= e$, changes s only on the bin $\delta(z)$ (if it exists), so the bins not in the images of δ and $\underline{\delta}$ are unaffected. Thus condition (D) in the definition clearly holds for the output states.

Let $e_1 := e^{[\vec{x} \to \vec{u}\ ,\ \vec{v} \to \vec{t}]}$. Then, by **8.9**, we have $e_1^{s \circ \delta} = e^{\underline{s} \circ \underline{\delta}}$. For this, note that $u_i \notin$ "e" for all i, since those variables cannot be free in K by CV_3, so the new condition (iii) in **8.9** holds.

Also, letting

$$z_1 \ := \ \begin{cases} u_i & \text{if } z = x_i\ ; \\ z & \text{if } z \notin \text{"}\vec{x}\text{"}; \end{cases}$$

we get

$$(z \div\!\!= e)^{[\vec{x} \to \vec{u}\ ,\ \vec{v} \to \vec{t}]} \ = \ z_1 \div\!\!= e_1\ .$$

(By CV_2, the v_i cannot appear left of the colon in an assignment command in K, so the case $z = v_i$ does not arise above.)

Here we will again use shorter notation for the left-hand side of state-pairs, viz.

$$\underline{\sigma}_{z \leftarrow e}(\underline{s}|\underline{\delta}) = \underline{\sigma}|\underline{\delta} \quad \text{and} \quad \sigma_{z \leftarrow e}(s|\delta) = \sigma|\delta$$

defines $\underline{\sigma}$ and σ.

Thus

$$\sigma := LOUT(z_1 \leftarrow\!\!\div\, e_1/\pi, s|\delta) = \begin{cases} \delta(u_i) \mapsto e_1^{s\circ\delta} = e^{\underline{s}\circ\underline{\delta}}\,, & \text{and} \\ \text{other bin}_q \mapsto s(\text{bin}_q) & \text{if } z = x_i; \\ \delta(z) \mapsto e_1^{s\circ\delta} = e^{\underline{s}\circ\underline{\delta}}\,, & \text{and} \\ \text{other bin}_q \mapsto s(\text{bin}_q) & \text{if } z \notin \text{“}\vec{x}\text{ ”}; \end{cases}$$

whereas

$$\underline{\sigma} := LOUT(z \leftarrow\!\!\div\, e/\pi, \underline{s}|\underline{\delta}) = \begin{cases} \underline{\delta}(z) \mapsto e^{\underline{s}\circ\underline{\delta}}\,, & \text{and} \\ \text{other bin}_q \mapsto \underline{s}(\text{bin}_q) & . \end{cases}$$

To check condition (C) for σ and $\underline{\sigma}$ in the “matching”-definition, we have $\underline{\sigma}\circ\underline{\delta}(v_j) = \underline{s}\circ\underline{\delta}(v_j) = t_j^{s\circ\delta} = t_j^{\sigma\circ\delta}$ as required, for the following reasons :

The first equality is immediate from the calculation just above of $\underline{\sigma}$, since $z = v_j$ cannot occur because none from $\vec{v}$ can appear to the left of the “:” in an assignment command by CV_2, so $v_j \neq z$.

The second equality is condition (C) for the “matching” assumed in this lemma.

The final equality is clear as long as $\sigma \circ \delta(y) = s \circ \delta(y)$ for all variables y in the term t_j. But that is clear from the calculation just above of σ, since no variable in t_j can be a u_i, or be z by CV_3.

To check condition (B), immediately from the basic calculations of $\underline{\sigma}$ and σ in the above large displays, we get

$$\sigma \circ \delta(u_k) = \begin{cases} e^{\underline{s}\circ\underline{\delta}}\,, & \text{if } z \notin \text{“}\vec{x}\text{ ”} \text{ and } u_k = z\,; \\ s \circ \delta(u_k) & \text{if } z \notin \text{“}\vec{x}\text{ ”} \text{ and } u_k \neq z\,; \\ e^{\underline{s}\circ\underline{\delta}}\,, & \text{if } z = x_k\,; \\ s \circ \delta(u_k) & \text{if } z = x_i \text{ and } i \neq k\,; \end{cases}$$

whereas

$$\underline{\sigma} \circ \underline{\delta}(x_k) = \begin{cases} e^{\underline{s}\circ\underline{\delta}}\,, & \text{if } z = x_k\,; \\ \underline{s} \circ \underline{\delta}(x_k) & \text{if } z \neq x_k\,. \end{cases}$$

But the very top of the six right-hand sides simply doesn't occur here, because of CV_3, viz. no variable in $\vec{u}$ is free in K. It is immediate that the two left-hand sides agree, as required, from condition (B) for the "matching" assumed in this lemma.

Finally, to check condition (A), assume that $y \notin$ "$\vec{x} \cup \vec{v} \cup \vec{u}$ ". Immediately from the early calculations of $\underline{\sigma}$ and σ, we find

$$\sigma \circ \delta(y) = \begin{cases} e^{\underline{s} \circ \underline{\delta}} & \text{if } z = x_i \text{ and } u_i = y \ ; \\ s \circ \delta(y) & \text{if } z = x_i \text{ and } u_i \neq y, \text{ so } y \neq z \text{ since } y \notin \text{“}\vec{x}\text{”} \ ; \\ e^{\underline{s} \circ \underline{\delta}} \ , & \text{if } z \notin \text{“}\vec{x}\text{”} \text{ and } y = z \ ; \\ s \circ \delta(y) & \text{if } z \notin \text{“}\vec{x}\text{”} \text{ and } y \neq z \ ; \end{cases}$$

whereas

$$\underline{\sigma} \circ \underline{\delta}(y) = \begin{cases} e^{\underline{s} \circ \underline{\delta}} \ , & \text{if } y = z \ ; \\ \underline{s} \circ \underline{\delta}(y) & \text{if } y \neq z \ . \end{cases}$$

If the first of the six right-hand sides cannot happen, it is immediate that the two left-hand sides agree, as required, using condition (A) for the "matching" assumed in this lemma. But the top right-hand side cannot happen because we picked a variable y with $y \notin$ "$\vec{u}$ ".

This finally completes the proof of **8.10**. The condition that $y \notin$ "$\vec{u}$ " in the final line of that proof is essential, as the following example shows. This is a counterexample to Lemma 3 in [**C**], where the definition of matching does not include that condition in its part (A). (Fortunately) the rest of Cook's proof goes through (because I don't know any different proof!) with the weaker definition of matching and the consequential weaker version of his Lemma 2, our **8.9**. Note that both **8.9** and **8.10** are used crucially several times in the proof of adequacy *and* in the proof of soundness.

For the example, take the variables to be $w_1, w_2, \cdots$; take $\underline{\delta}$ and δ both to be defined on w_1, w_2, w_3 only, mapping w_i to bin_i, except that $\underline{\delta}$ maps w_1 to bin_2 and w_2 to bin_1; take $\underline{s} = s$ to be the state given by $0, 0, 1, 0, 0, 0, \cdots$; take K to be the assignment command $w_1 \Leftarrow w_3$; and finally take $\vec{v}$ and $\vec{t}$ to be empty, and $\vec{x} = (w_1)$ and $\vec{u} = (w_2)$, i.e. $u_1 = w_2$.

This satisfies the hypotheses of **8.10** even if we don't require $y \notin$ "$\vec{u}$ " in (A) of the definition of matching, i.e. if we'd used Cook's definition. Our definition requires $\underline{\sigma} \circ \underline{\delta}(y) = \sigma \circ \delta(y)$ only for $y = w_3$, whereas the stronger version of (A) requires it also for $y = w_2$. But both hold, w_2 giving the value 0, and w_3 the value 1.

A quick calculation gives $\underline{\sigma} = \sigma$ to be the state given by $0, 1, 1, 0, 0, 0, \cdots$. The substitution changes the command to $w_2 \Leftarrow w_3$. Then the matching conclusion of **8.10** of course holds for our 'matching definition'. But it fails for the other, because

$$0 = \underline{\sigma} \circ \underline{\delta}(w_2) \neq \sigma \circ \delta(w_2) = 1 \ .$$

Continuation of the proof of 8.7.
The final case for adequacy is for call commands.
Assume that $F\{\mathsf{call}\ p(\vec{u} : \vec{t})/\pi\}G$ is true, and we'll find a derivation.

The asinine subcase is that when p is not in the domain of π. (In that case, F and G could be any formulas.) Extend π to π', which agrees with π, and has only p added to its domain, with $\pi'(p) = B$, where B is some command which fails to terminate no matter what the input. By (XI), it suffices to find a derivation for $F\{\mathsf{call}\ p(\vec{u} : \vec{t})/\pi\}G$. By (VIII), it suffices to find a derivation for $F\{B/\pi'\}G$. The analogue was done just before the proof of **8.4** when we were in the child's world of the language **ATEN**. The solution here does require going through all seven cases in the structural inductive definition of commands in **DTEN**. But it is essentially elementary and will be left to the reader as an exercise. (Note that π might just as well be empty here.)

In the non-asinine subcase, let $\pi(p) = (K, (\vec{x} : \vec{v}))$. Let $\vec{z}$ be a list of all variables in $\vec{t}$. Let $\vec{x'}, \vec{v'}, \vec{x''}, \vec{v''}, \vec{z'}$ be a string of distinct variables, completely disjoint from any variables occurring so far in this case, and any variables in $FREE(\mathsf{call}\ p(\vec{u} : \vec{t})/\pi)$. (The individual string lengths can be guessed from the letters used, or deduced from the various substitutions below.)

Let $\vec{t'} := \vec{t}^{\,[\vec{z} \to \vec{z'}]}$, and define

$$F_1 \ := \ \vec{v} \approx \vec{t'} \ \wedge \ \left(F^{[(\vec{x},\vec{v}) \to (\vec{x''},\vec{v''})]}\right)^{[\vec{u} \to \vec{x}\ ,\ \vec{z} \to \vec{z'}]} .$$

The notation used should be fairly obvious. In particular, the 'vector' equality formula before the conjunction is really a *multiple conjunction* of equalities of variables with terms. Get G_1 from G by the same pair of two successive simultaneous substitutions that were applied to F in the display just above :

$$G_1 \ := \ \left(G^{[(\vec{x},\vec{v}) \to (\vec{x''},\vec{v''})]}\right)^{[\vec{u} \to \vec{x}\ ,\ \vec{z} \to \vec{z'}]} .$$

Lemma 8.11 *The F-H statement $F_1\{K/\pi'\}G_1$ is true, where π' agrees with π, except that p is not in the domain of π'.*

Delaying the proof, it now follows by the inductive hypothesis pertaining to π' being 'smaller' than π that the F-H statement in the lemma is derivable.

Immediately from rule (VIII), the statement

$$F_1\{\mathsf{call}\ p(\vec{x} : \vec{v})\ /\ \pi\}G_1$$

is derivable.

Now, in rule (X), take $\vec{y}, \vec{w}, \vec{u}, \vec{t}$ to be $\vec{x}, \vec{v}, \vec{x'}, \vec{v'}$ respectively. (So we are temporarily changing $\vec{u}, \vec{t}$ from its fixed meaning in this case.) Certainly no x'_i even occurs in F_1, so the rule applies. Looking at the definition of F_1, (and noting that $[\vec{u} \overset{(i)}{\longmapsto} \vec{x} \overset{(ii)}{\longmapsto} \vec{x'}$ and $\vec{v} \overset{(ii)}{\longmapsto} \vec{v'}$, where (ii) is the substitution in the lower line of rule (X), and (i) is in the superscript in the definition of F_1), we get the statement as follows to be derivable:

$$(\vec{v'} \approx \vec{t'} \wedge (F^{[(\vec{x},\vec{v})\mapsto(\vec{x''},\vec{v''})]})^{[\vec{u}\mapsto\vec{x'}\ ,\ \vec{z}\mapsto\vec{z'}]})\ \{\mathsf{call}\ p(\vec{x'} : \vec{v'})/\pi\}\ (G^{[(\vec{x},\vec{v})\mapsto(\vec{x''},\vec{v''})]})^{[\vec{u}\mapsto\vec{x'}\ ,\ \vec{z}\mapsto\vec{z'}]}\ .$$

Now easily apply rule (IX) with $\vec{y} = \vec{z'}$, $\vec{r} = \vec{z}$ and $C = \mathsf{call}\ p(\vec{x'} : \vec{v'})$ to get the same thing as above, except that the "$\vec{t'}$" becomes "$\vec{t}$" on the far left, and the ", $\vec{z}\mapsto \vec{z'}$ " are erased on the far right of the superscripts for both F and G.

Again, in rule (X), take $\vec{y}, \vec{w}, \vec{u}, \vec{t}$ to be $\vec{x'}, \vec{v'}, \vec{u}, \vec{t}$ respectively, (where we are now back to the fixed meaning of $\vec{u}, \vec{t}$). Certainly, no u_i occurs free in the pre- and post- conditions of the F-H statement because of the $\vec{u}\mapsto \vec{x'}$. So (X) does apply, and the substitutions in its lower line are $\vec{x'}\mapsto \vec{u}$, $\vec{v'}\mapsto \vec{t}$. So we get the statement as follows to be derivable:

$$(\vec{t} \approx \vec{t}\ \ \wedge\ \ F^{[(\vec{x},\vec{v})\mapsto(\vec{x''},\vec{v''})]})\ \{\mathsf{call}\ p(\vec{u} : \vec{t}\ )/\pi\}\ G^{[(\vec{x},\vec{v})\mapsto(\vec{x''},\vec{v''})]}\ .$$

Now we can just erase the "$\vec{t} \approx \vec{t}\ \wedge$" at the front, by using rule (IV) with the upper line

$$F_2\ \rightarrow\ \vec{t} \approx \vec{t}\ \wedge F_2\quad ,\quad (\vec{t} \approx \vec{t}\ \wedge F_2)\{C/\pi\}G_2\quad ,\quad G_2 \rightarrow G_2\quad ,$$

where the middle of the display is the F-H statement in the display second above.

Finally, in rule (IX), take $\vec{y} = (\vec{x''}, \vec{v''})$, $\vec{r} = (\vec{x}, \vec{v})$ and $C = \mathsf{call}\ p(\vec{u} : \vec{t})$ to obtain, as required, the derivability of $F\{\mathsf{call}\ p(\vec{u} : \vec{t}\ )/\pi\}G$. The rule is applicable, since variables in $(\vec{x''}, \vec{v''})$ are not in $FREE(\mathsf{call}\ p(\vec{u} : \vec{t}\ ))$ by the choice of those new variables disjoint from earlier ones, and those in $(\vec{x}, \vec{v})$ also do not appear in $FREE(\mathsf{call}\ p(\vec{u} : \vec{t}\ )/\pi)$ by its definition and our caveat CV_4+ .

This completes the proof of the adequacy half of **8.7**, modulo proving **8.11**, which we now proceed to do, before going on to proving the soundness half of **8.7**. Notice that the only two uses of rule (IX) above had $\vec{r}$ as a list of variables both times and also had the command as a call-command both times. So, at least to that extent, the proof system could be pared to a minimum by restricting (IX) to that situation.

Proof of Lemma 8.11.

By the definition of SSQ, the truth of $F\{\mathsf{call}\ p(\vec{u} : \vec{t}\,)/\pi\}G$ (assumed for the final case of adequacy) gives the truth of $F\{K^{[\vec{x}\to\vec{u}\ ,\ \vec{v}\to\vec{t}]}\ /\ \pi'\}G$. That in turn gives the truth of

$$F^{[(\vec{x},\vec{v})\to(\vec{x''},\vec{v''})]}\{K^{[\vec{x}\to\vec{u}\ ,\ \vec{v}\to\vec{t}]}\ /\ \pi'\}G^{[(\vec{x},\vec{v})\to(\vec{x''},\vec{v''})]} \qquad (*)$$

because none of the variables involved in the formula substitutions occur in the command.

Now assume F_1 to be true at $s_1 \circ \delta_1$, where δ_1 is defined at least on all free variables in F_1 and G_1 and K. Also assume that $SSQ(K/\pi', s_1|\delta_1)$ is a finite sequence. We must show that G_1 is true at $OUT(K/\pi', s_1|\delta_1)$.

Pick an $s|\delta$ so that the following four conditions hold:

$$s \circ \delta(y) \ = \ s_1 \circ \delta_1(y) \ \text{ for all } y \notin \vec{x} \cup \vec{v} \text{ such that } \delta_1(y) \text{ exists ;}$$

$$s \circ \delta(u_i) \ = \ s_1 \circ \delta_1(x_i) \ ;$$

$$s \circ \delta(z) \ = \ s_1 \circ \delta_1(z') \ ;$$

$$s_1(\mathrm{bin}_{i_1}) = s(\mathrm{bin}_i) \text{ for } i_1 - m_1 = i - m \geq 0 \text{ where } \ m_1 \text{ and } m$$

are the smallest subscripts of bins larger than the subscripts of all bins in the images of δ_1 , δ respectively.

The pair $s_1|\delta_1$ is then $(\vec{x}\to\vec{u}\ ;\ \vec{v}\to\vec{t}\,)$—matched to $s|\delta$. Checking condition(C) in the definition of "—matched" also uses the "$\vec{v} \approx \vec{t}$-part" of F_1 .

Because of the definition of (s, δ) on the variables in $F^{[(\vec{x},\vec{v})\to(\vec{x''},\vec{v''})]}$, that formula is true at $s \circ \delta$ because (the right-hand half of) F_1 is true at $s_1 \circ \delta_1$.

Combining this with $(*)$, we see that $G^{[(\vec{x},\vec{v})\to(\vec{x''},\vec{v''})]}$ is true at

$$OUT(K^{[\vec{x}\to\vec{u}\ ,\ \vec{v}\to\vec{t}]}/\pi', s|\delta) \ = \ OUT(K^{[\vec{x}\to\vec{u}\ ,\ \vec{v}\to\vec{t}]}/\pi, s|\delta) \ .$$

But, by **8.10**, the OUT in this display is $(\vec{x}\mapsto \vec{u}\ ;\ \vec{v}\mapsto \vec{t})$—matched to

$$OUT(K/\pi', s_1|\delta_1) \ := \ OUT(K/\pi, s_1|\delta_1)\ ,$$

Changing from π' to π in both OUT-displays above follows from **8.6**d) just below, since no calls to p can occur in the commands, by CV_1.

Combining these facts from the OUT-displays with **8.9**, we find that G_1 is true at $OUT(K/\pi', s_1|\delta_1)$, as required.

8.6(d) *Let* $C \in$**DTEN** *and let* π', π *and* p *be such that*
(i) $p \notin \mathrm{dom}(\pi')$ *;*
(ii) $\mathrm{dom}(\pi) = \{p\} \cup \mathrm{dom}(\pi')$ *and* π, π' *agree except that* $\pi(p)$ *is defined ;*
(iii) *The procedure identifier* p *is not in any indirect activation sequence arising from* C/π *.*
Then

$$SSQ(C/\pi, s|\delta) \ = \ SSQ(C/\pi', s|\delta) \quad \text{for any } s \text{ and } \delta\ .$$

The proof is a straightforward induction.

Completion of the proof 8.7 (i.e. proof of the soundness half).
The validity of rules (I) to (IV) is very straightforward, with proof similar to the arguments given in tedious detail in **8.1** for **ATEN**, so these four will be omitted here.

Rule (V) is beyond the pale of triviality.

Validity of rule (VI) :

Let $C = \mathsf{begin}\ D_*; C_*\ \mathsf{end}$ and $C_+ = \mathsf{begin\ new}\ x; D_*; C_*\ \mathsf{end}$. Assume the truth of the upper line in (VI), i.e.

$$F^{[x\to y]}\ \{C/\pi\}\ G^{[x\to y]} \qquad \text{is true} \qquad\qquad (*)$$

To show the truth of the lower line, i.e. $F\ \{C_+/\pi\}\ G$, assume that F is true at $s \circ \delta$, and that $SSQ(C_+/\pi, s|\delta)$ is a finite sequence.

We must prove that G is true at $OUT(C_+/\pi, s|\delta)$.

This comes from

F true at $s \circ \delta \Rightarrow F^{[x \to y]}$ true at $s \circ \delta^* \Rightarrow G^{[x \to y]}$ true at $OUT(C/\pi, s|\delta^*)$

$$\Rightarrow G \text{ true at } LOUT(C/\pi, s|\delta^*) \mid \delta'' \Rightarrow G \text{ true at } OUT(C_+/\pi, s|\delta) \,,$$

where we define δ' and δ'' as in the definition of SSQ for variable-declaration-commands, and define $\delta^* = \delta'$ except that $\delta^*(y) = \delta(x)$.

The first implication is because δ and δ^* agree, except that $\delta^*(y) = \delta(x)$ and the values $\delta^*(x)$ and $\delta(y)$ (if defined) are irrelevant, because y is not free in F, and x is not free in $F^{[x \to y]}$.

The second implication is immediate from $(*)$.

The third implication follows because δ'' and $ROUT(C/\pi, s|\delta^*)$ agree, except the first maps x to what the second maps y to, and their values on y and x respectively are irrelevant.

The fourth implication follows from

$$OUT(C_+/\pi, s|\delta) \;=\; LOUT(C/\pi, s|\delta') \mid \delta'' \;=\; LOUT(C/\pi, s|\delta^*) \mid \delta'' \,,$$

where the second equality holds because δ^* and δ' agree except on y, but y is not in $FREE(C/\pi)$. (This may be proved directly in this simple case, but it is also the one application for **8.8**.) The first equality is immediate from the definition of $SSQ(C_+/\pi, s|\delta)$ in terms of $SSQ(C/\pi, s|\delta')$.

Validity of rule (VII) :

Let $C = \mathsf{begin}\ D_*; C_*\ \mathsf{end}$ and $C_+ = \mathsf{begin}\ D; D_*; C_*\ \mathsf{end}$, where $D = \mathsf{proc}\ p(\vec{x} : \vec{v}) \equiv K\ \mathsf{corp}$. Assume the truth of the upper line in (VII), i.e. $F\ \{C/\pi'\}\ G$ is true . To show, as required, that $F\ \{C_+/\pi\}\ G$ is true , where $\pi = \pi'$ except that $\pi'(p) = (K, (\vec{x} : \vec{v}))$, suppose that F is true at $s \circ \delta$, and that $SSQ(C_+/\pi, s|\delta)$ is a finite sequence. We must check that G is true at $OUT(C_+/\pi, s|\delta)$ But it is true at $OUT(C/\pi', s|\delta)$, so we only need the equality of these "OUTs". And that follows because the sequences $SSQ(C_+/\pi, s|\delta)$ and $SSQ(C/\pi', s|\delta)$, by definiton of SSQ, only differ by a spurious initial term.

Validity of rule (VIII) :

Let $C_+ = \mathsf{call}\ p(\vec{x} : \vec{v})$ and suppose that $F\{C/\pi'\}G$, the upper line in this rule, is true, where $\pi(p) := (C, (\vec{x} : \vec{v}))$ adds p to the domain to give π with a strictly larger domain than π'. To show that $F\{C_+/\pi\}G$, the lower line, is true, suppose that F is true at $s \circ \delta$, and that $SSQ(C_+/\pi, s|\delta)$ is a finite sequence.

To show, as required, that G is true at $(OUT(C_+/\pi, s|\delta)$, note that G is true at $OUT(C/\pi', s|\delta)$, which exists because of the display below. We claim

$$SSQ(C_+/\pi, s|\delta) \ = \ \prec\ s\ ,\ SSQ(C^{[\ \vec{x}\to\vec{x}\ ,\ \vec{v}\to\vec{v}\ ]}\ /\ \pi, s|\delta)\ \succ$$

$$= \ \prec\ s\ ,\ SSQ(C/\pi, s|\delta)\ \succ \ = \ \prec\ s\ ,\ SSQ(C/\pi', s|\delta)\ \succ\ .$$

And so the two relevant OUTs agree, and we're done, as long as the equalities claimed above are correct. The first equality is just the definition of SSQ for C_+ in terms of that for C. The substitution is a 'non-starter', giving the second one. Using **8.6**(d), the last equality holds since CV_1 implies (iii) in **8.6**(d) for C/π.

Validity of rule (IX) :

Assume that $F\{C/\pi\}G$ is true, and that no variable y_i nor any variable in any term r_i is in $FREE(C/\pi)$. To show, as required, that $F^{[\ \vec{y}\to\vec{r}\ ]}\{C/\pi\}G^{[\ \vec{y}\to\vec{r}\ ]}$ is also true, assume that $F^{[\ \vec{y}\to\vec{r}\ ]}$ is true at $s \circ \delta$, and that $SSQ(C/\pi, s|\delta)$ is a finite sequence. We must prove that $G^{[\ \vec{y}\to\vec{r}\ ]}$ is true at $(OUT(C/\pi, s|\delta)$.

Determine s_1 by

$$s_1 \circ \delta(z) \ = \ s \circ \delta(z) \ \text{ for } z \notin \text{“}\vec{y}\,\text{”} \text{ with } z \in \mathrm{dom}(\delta)\ ;$$

$$s_1 \circ \delta(y_i) \ = \ r_i^{s\circ\delta} \ ;$$

$$s_1(\mathrm{bin}_n) = s(\mathrm{bin}_n) \text{ for } \mathrm{bin}_n \notin \mathrm{Image}(\delta)\ .$$

Then $s_1|\delta$ is (empty subn. , $\vec{y}\to\vec{r}$)—matched to $s|\delta$. In twice applying **8.9** below, hypothesis (iii) there holds vacuously.

By **8.9**, F is true at $s_1 \circ \delta$, and so, since $F\{C/\pi\}G$ is true, G is true at $OUT(C/\pi, s_1|\delta)$.

By **8.10**, $OUT(C/\pi, s_1|\delta)$ is (empty subn. , $\vec{y}\to\vec{r}$)—matched to $OUT(C^{[\ \vec{y}\to\vec{r}\ ]}/\pi, s|\delta)$.

Thus, by **8.9**, $G^{[\ \vec{y}\to\vec{r}\ ]}$ is true at $OUT(C^{[\ \vec{y}\to\vec{r}\ ]}/\pi, s|\delta)$.

Now the following general result gives that last OUT to agree with $OUT(C/\pi, s|\delta)$, so we're done.

8.6(e) *If no y_i nor any variable in r_i is in $FREE(C/\pi)$, then $C^{[\ \vec{y}\to\vec{r}\ ]} = C$*

.

The proof of this almost tautological fact is a straightforward induction.

Validity of rule (X) :

Let $C = \mathsf{call}\ p(\vec{y} : \vec{w})$ and $C_+ = \mathsf{call}\ p(\vec{u} : \vec{t})$, with conditions as in rule (X) holding.

If $\pi(p)$ is not defined, there is nothing to prove, since then, $F_1\{C_+/\pi\}G_1$ is true for any F_1 and G_1, because $SSQ(C_+/\pi, -|-)$ is undefined..

So let $\pi(p) = (K, (\vec{x}, \vec{v}))$. Suppose that $F\{C/\pi\}G$ is true. To show that

$$F^{[\ \vec{y}\to\vec{u}\ ,\ \vec{w}\to\vec{t}\ ]}\{C_+/\pi\}G^{[\ \vec{y}\to\vec{u}\ ,\ \vec{w}\to\vec{t}\ ]}$$

also is true, as required, assume that $F^{[\ \vec{y}\to\vec{u}\ ,\ \vec{w}\to\vec{t}\ ]}$ is true at $s \circ \delta$, and that $SSQ(C_+/\pi, s|\delta)$ is a finite sequence.

We must prove that $G^{[\ \vec{y}\to\vec{u}\ ,\ \vec{w}\to\vec{t}\ ]}$ is true at $OUT(C_+/\pi, s|\delta)$.

Pick some $s_1|\delta_1$ which is $(\vec{y}\to\ \vec{u}, \vec{w}\to\ \vec{t})$—matched to $s|\delta$. Then F is true at $s_1 \circ \delta_1$ by **8.9**. And so G is true at $OUT(C/\pi, s_1|\delta_1)$, since $F\{C/\pi\}G$ is true.

Because of the way SSQ is defined for call-commands,

$$OUT(C_+/\pi, s|\delta)\ =\ OUT(K^{[\ \vec{x}\to\vec{u}\ ,\ \vec{v}\to\vec{t}\ ]}/\pi\ ,\ s|\delta)\ .$$

Define $L\ :=\ K^{[\ \vec{x}\to\vec{y}\ ,\ \vec{v}\to\vec{w}\ ]}$, so that

$$K^{[\ \vec{x}\to\vec{u}\ ,\ \vec{v}\to\vec{t}\ ]}\ =\ L^{[\ \vec{y}\to\vec{u}\ ,\ \vec{w}\to\vec{t}\ ]}\ .$$

But this last equality gives that

$$OUT(C_+/\pi, s|\delta)\ =\ OUT(L^{[\ \vec{y}\to\vec{u}\ ,\ \vec{w}\to\vec{t}\ ]}/\pi\ ,\ s|\delta)\ .$$

Now

$$\begin{aligned} OUT(C/\pi, s_1|\delta_1)\ &:=\ OUT(\mathsf{call}\ p(\vec{y} : \vec{w})/\pi, s_1|\delta_1) \\ &=\ OUT(K^{[\ \vec{x}\to\vec{y}\ ,\ \vec{v}\to\vec{w}\ ]}/\pi\ ,\ s_1|\delta_1)\ :=\ OUT(L/\pi, s_1|\delta_1)\ . \end{aligned}$$

The non-definitional equality is immediate from the definition of SSQ for call-commands.

By **8.10**, the right-hand side OUTs of the previous two displays are matched, and so the left-hand sides also are; that is, $OUT(C/\pi, s_1|\delta_1)$ is $(\vec{y}\!\!\rightarrow \vec{u}, \vec{w}\!\!\rightarrow \vec{t})$—matched to $OUT(C_+/\pi, s|\delta)$.

Using **8.9**, and the truth of G at $OUT(C/\pi, s_1|\delta_1)$, we see that $G^{[\ \vec{y}\rightarrow\vec{u}\ ;\ \vec{w}\rightarrow\vec{t}\ ]}$ is true at $OUT(C_+/\pi, s|\delta)$, as required.

Validity of rule (XI) :

Here we use a final general result about the semantics, whose proof is yet another straightforward induction.

8.6(f) *If $\pi \subset \pi'$ and $SSQ(C/\pi, s|\delta)$ is a finite sequence, then so is $SSQ(C/\pi', s|\delta)$, and they have the same last term.*

Now assume that $F\{C/\pi'\}G$ is true, and that F is true at $s \circ \delta$. Suppose that $SSQ(C/\pi, s|\delta)$ is a finite sequence. The result above gives that $SSQ(C/\pi', s|\delta)$ is also a finite sequence, and that

$$OUT(C/\pi', s|\delta) \ = \ OUT(C/\pi, s|\delta) \ .$$

So G is true at $OUT(C/\pi, s|\delta)$, as required to show that $F\{C/\pi\}G$ is also true.

8.4—The non-existence of a complete F-H proof system for the full languages ALGOL and PASCAL.

The language **CTEN** is nowhere close to full ALGOL. Only a few of the caveats apply there. For example, certainly recursive procedures are allowed, indeed, very much encouraged! (But see Addendum 3 below.) And there are function declarations, GOTO-commands, etc. in ALGOL. So the following question arises: for which practical software languages do we have the existence of a complete F-H proof system in the sense of the theorems of the last two subsections?

In [**Cl1**] there are a number of cases more general than **CTEN** of this last theorem where such a proof system is claimed to exist, and a partial proof is given in one of these cases. The literature does abound with papers on this sort of question. But many of them also refer, more-or-less explicitly, to mistakes in earlier papers. The situation can be mildly confusing to a neophyte with mathematical interests but not a lot of knowledge of the CS jargon nor of which claims are to be regarded as reliable.

Here are a few of the negative quotes:

> In [**Cl1**] it was argued that if use of global variables was disallowed, then denesting of internal procedures would be possible. Thus, the proof system ... could be adapted.... This argument was shown to be incorrect by Olderog.
>
> Clarke 1984[**Hoare-Shepherdson eds**]p.98

An amusing sidenote here is that, in the title in Clarke's reference list for his own paper (our [**Cl1**]), the word "impossible" got changed to "possible"!

> We thank ... G.Plotkin, for his elegant counterexamples to the soundness of Clarke's procedure call axiom.
>
> Trakhtenbrot, Halpern,Mayer 1983[**Clarke-Kozen eds**]p.497

> Whether such systems are to be used for formal verification, by hand or automatically, or as a rigorous foundation for informal reasoning, it is essential that they be logically sound. Several popular rules in the Hoare logic are in fact not sound. These rules have been accepted because they have not been subjected to sufficiently strong standards of correctness.
>
> M.J. O'Donnell 1981[**Kozen-ed.**]p.349

> The rules for procedure call statements often (in fact usually) have technical bugs when stated in the literature, and the rules stated in earlier versions of the present paper are not exceptions.
>
> S. Cook 1978[**C**]p.70

One major motivation for the (excessive?) detail in the last subsection is just this queasy situation. I hope that I got it right! As mentioned earlier, the paper [**Old1**] gives fairly comprehensive results, and seems to be unquestioned as far as accuracy is concerned, though some disagree with the specification method for the semantics. It deals essentially with producing careful analysis of F-H proof systems for the five cases of command languages with procedure declarations plus all but one of the five language features displayed below in italics, where we explain briefly why such a system couldn't exist with all five features in the language. In the third addendum below, we at least give a system for a language with recursive programming.

Note also that O'Donnell's remarks quoted above concentrate on two features of programming languages which have not been discussed at all here yet—*declared functions* and *GOTO-commands*. It is interesting that 'Frege's Folly', AKA Russell's Paradox, was inadvertantly re-created 90 years after Frege in trying to invent proof rules for function declarations, as first observed by Ed Ashcroft, an occurrence which undoubtedly helped to get people serious about proving soundness of proof rules in this subject. See Addendum 1 below for a short discussion on function declarations and Ashcroft's observation, and see O'Donnell's paper quoted above for other details.

I don't know whether there is anything close to a rigorously proved complete F-H proof system for an ALGOL-style language such that any 'known' language feature which is missing cannot be added without making it impossible for such a proof system to exist. Most work after the mid-80's seems to concentrate on parallel and concurrent programming.

So we'll finish with a brief description of the major *negative* result in [**Cl1**], which apparently implies that there cannot be a complete F-H proof system for full ALGOL. It is natural to wonder whether, and to what extent, all this theory influenced the design of C, C++, C#, etc.

No attempt will be made to give exhaustive mathematical details here. The negative argument of the first subsection gives us the following. Suppose given a command language $\mathcal{C}$ which has the $1^{\underline{\text{st}}}$order language $\mathcal{L}$ as its basis. Fix an interpretation I of $\mathcal{L}$, so that $\mathcal{C}$ can be given a corresponding semantics. Assume this is done with sufficient rigour that one can prove:

(1) the halting problem for $\mathcal{C}$ relative to I to be undecidable; and

(2) there is a complete axiomatic proof system for $\mathcal{L}$ relative to I.

It follows from (1) that the set of commands C which loop on all inputs is not recursively enumerable. But a complete F-H proof system plus (2) would imply that set to be recursively enumerable.

Now (2) will certainly hold if I is finite. Clarke's major achievement here was to show how, if $\mathcal{C}$ had a certain list of features, namely:

(if you know the jargon) —*procedures as parameters of procedure calls, recursion, static scope, global variables, and internal procedures*—

then, for every I with more than one element (including of course *finite* I), the halting problem for $\mathcal{C}$ is undecidable, so (1) holds. And therefore the standard argument of Cook above applies : *there can be no (relatively) complete F-H proof system for such a language.*

It hardly needs saying that this surprising theorem of Clarke has been of huge importance in the field.

Addendum 1 : Function Declarations and Ashcroft's 'Paradox'.

We introduce a very simple function declaration string, to jazz up the language in Subsection 1. Funnily enough, it takes the form

$$\mathsf{fun}\ y \sim f(x) \Leftarrow K\ \mathsf{nuf}\quad ,$$

where K is a command (very likely containing at least the variable x). Beforehand, one has specified a set $FIDE$ of function identifiers with typical member f. The above y is then intuitively "the value of the function f at x", where y really refers to the value of that variable in the state *after* K has been executed, and x really to its own value in the state *before* K is executed.

A function declaration is not dissimilar to a procedure declaration. But one is, so to speak, only interested in the one value, y, within the output from K, not with the entire output state. And there is no direct function **call**—$f(\cdots)$ is simply allowed to be used wherever we formerly used terms from the $1^{\underline{\text{st}}}$order language, anywhere within the block starting with the function declaration, in assertions, commands and other declarations. So, for example,

$$\mathbf{whdo}(f(x) + z < w \times f(t + z))(C)$$

and

$$w \dot{\leftarrow} f(x) + z$$

would be a new sorts of commands, usable within that block; and

$$\forall w\ (f(x) + z < w \times f(t + z))$$

would be a new sort of assertion, for any variables x, z, w and any term t.

Thus one would define three sets, say ASS, COM and DEC, of assertions ("formulas"), commands and declarations, respectively, by simultaneous structural induction. We won't go into details here. For the example below, you can imagine that DEC has only function (not variable nor procedure) declarations; that COM is the usual **ATEN** except for also using blocks beginning with a declaration, and allowing the use of f as just above in assignments and in the formulas for controlling 'while-do'-commands; and that ASS is just '$1^{\underline{\text{st}}}$order number theory souped up with functions', again using f as just above, and also in substitutions for variables.

One sanitary requirement would be that any $f \in FIDE$ occurs at most once in declarations within a block. Also, we might as well forbid recursive function declarations for our simple purposes here.

A proper treatment would give also a careful definition of the semantics, depending on a chosen interpretation of the $1^{\underline{\text{st}}}$order language. Again what's below is independent of the interpretation, but if definiteness is desired, take it to be **N**.

Now a rule which had been proposed, to go along with the four usual rules for **ATEN**, is

$$\frac{F\{K\}G}{F \to G^{[y \to f(x)]}} \quad ,$$

for any F, G in ASS, and K in COM, where f is a function identifier appearing in a declaration as at the beginning of the addendum. (We can weaken it to stick to $1^{\underline{\text{st}}}$order formulas F and G without defined functions, but the Ashcroft example below still applies.)

The reader is urged not to think too much about this rule right now, other than to say quietly: *"That looks reasonable—when F is true then G with y replaced by $f(x)$ must be true, if f is computed using K, and the F-H statement in the numerator is true"*. If you feel a bit puzzled about this rule, after the Ashcroft contradiction just below, our discussion will hopefully explain away any puzzlement.

Now let K be the command $\mathbf{whdo}(0 \approx 0)(x \doteqdot x)$, which for these purposes could be any command which just loops forever on any input. Let f occur in the declaration as at the beginning with this particular K.

Now we get a derivation as follows :

$$\begin{array}{l} \quad\bullet \\ \quad\bullet \\ \quad\bullet \\ 0 \approx 0\{K\}\neg 0 \approx 0 \\ 0 \approx 0 \to (\neg 0 \approx 0)^{[y \to f(x)]} \\ \quad\bullet \\ \quad\bullet \\ \neg 0 \approx 0 \end{array}$$

The first few lines would be a derivation fragment as discussed in (iii) after the statement of **8.4** ; because K loops, one could have any pre- and post-conditions here. The next line just applies the new rule. And the last

few lines merely give a $1^{\underline{\text{st}}}$order derivation, using the logical validity of $0 \approx 0$, and a presumed rule which at least allows us to derive H from $H^{[y \to f(x)]}$ when the formula H has no free y. Actually H and $H^{[y \to f(x)]}$ must be exactly the same string under these circumstances, once all the definitions have been fleshed out in detail.

Being able to derive $\neg 0 \approx 0$ seems undesirable, to say the least, and we'll just leave it at that—there seems little motivation at this point to try to find any kind of semantics which are reasonable, and for which the given rule is sound!

Additional Remarks. At first, having a rule whose conclusion is like a $1^{\underline{\text{st}}}$order formula seems puzzling. Heretofore, each rule had an F-H statement as its conclusion. After all, if we are working 'oracularly' as discussed in Subsection 1, we are **given** all true $1^{\underline{\text{st}}}$order formulas by the oracle, so who needs to derive them?

Also the rule's conclusion isn't actually in the $1^{\underline{\text{st}}}$order language because of the occurrence of f.

These two objections seem to cancel each other : the conclusion *is* in the enlarged $ASS \supset FORM$, and maybe we want to be able to derive 'formulas' in ASS of the form $H \to H'$, for use in a souped up version of the rule

$$\text{(IV)} \qquad \frac{F \to F' \ , \ F'\{C\}G' \ , \ G' \to G}{F\{C\}G}$$

This unfortunate 'rule' was originally stated with a universally quantified conclusion $\forall x \ (F \to G^{[y \to f(x)]})$. However, the presence or absence of the quantifier is moot, if one is actually talking about truth in an interpretation, as opposed to truth at a particular state.

But now one ponders it a bit more and puzzles concerning the restriction in the rule about f appearing in a certain declaration. This is getting stranger, since after all, given any two $1^{\underline{\text{st}}}$order formulas F and G, we can certainly find a command K already in the original **ATEN** for which the numerator of this rule can be derived. That's just the discussion of (iii) after the statement of **8.4** again. So the 'rule' would allow one to derive *any* assertion as in its denominator. And it seems to have no relevance at all to F-H logic as it has been presented here, since f doesn't really mean anything in particular.

One suspects that the intent of this rule was for producing intermediate $1^{\underline{\text{st}}}$order formulas somewhere 'in the middle of an execution of a program'. These formulas were presumably supposed to be true for the state produced at that stage. The intended rule is perhaps more like this :

$$\frac{F\{K\}G}{Tr\{\mathsf{begin\ fun}\ y \sim f(x) \Leftarrow K\ \mathsf{nuf}\ ;\ Null\ \mathsf{end}\}(F \rightarrow G^{[y \rightarrow f(x)]})} \quad ,$$

where Tr and $Null$ are any formula and command which are logically valid and 'do nothing', respectively.

In any case, Ashcroft's contradiction above applies just as well in that context, so it was "back to the drawing-board". General discussions, building up from 'while' to more complicated command-constructions, of F-H logic, such as [**Apt**] or [**deB**], seem to carefully avoid this topic, perhaps for good reason. Other than deBakker-Klop-Meyer, p.94 in [**Kozen-ed.**], I haven't been able to find any thorough analysis of function declarations for F-H logic as it is normally presented. That paper doesn't include detailed proofs of soundness and completeness. It is interesting that the set-up there is quite elaborate, involving a functional programming language to supplement the command language, and 'meaning' definitions which are given using denotational semantics.

Addendum 2 : Propositional Connectives applied to F-H Statements, and Total Correctness.

There are two aspects of the formulation of F-H logic earlier which some may find unnatural because of a lack of generality :

(1) The F-H statement $F\{C\}G$ (or $F\{C/\pi\}G$) is like a *closed* $1^{\underline{\text{st}}}$order formula in that it is either true or false in an interpretation: its truth value does not vary with different states from a given interpretation. It may seem more natural to introduce a syntactic notation $[F : C : G]$ whose semantics will say that it is true at a given state $\underline{v}$ exactly when [*if* F *is true at* $\underline{v}$ *and* $||C||(\underline{v}) \neq err$, *then* G *is true at* $||C||(\underline{v})$]. In this addendum, we shall always have C from a command language that is called the 'while' language, basically **ATEN** plus the if-then-else command construction (a halfway house between **ATEN** and **BTEN**). So the semantics is self-evident. One can think of the string $F\{C\}G$ as the (universal) closure of $[F : C : G]$.

(2) The other 'unnaturality' is in only dealing with F-H statements themselves, and not, say, with negations or conjunctions of them, or implications between them. As illustrated in the next addendum, when trying to give an F-H proof system for a command language with procedures in which *recursive* programs are allowed, it is almost *necessary* to deal not just with F-H statements, but also with implications and conjunctions iteratively applied to them, such as

$$F\{C\}G \;\&\; F'\{C'\}G' \qquad \text{or} \qquad (\; F\{C\}G \implies F'\{C'\}G' \;) \;\&\; F''\{C''\}G'' \quad .$$

Temporarily, we shall use

$$\neg \;\; ; \;\; \& \;\; ; \;\; \implies \;\; ; \;\; \iff \;\; ,$$

to keep these connectives distinct from the analogous connectives

$$\neg \;\; ; \;\; \wedge \;\; ; \;\; \longrightarrow \;\; ; \;\; \longleftrightarrow \;\; ,$$

which are parts of formulae from $\mathcal{L}$, the underlying $1^{\underline{\text{st}}}$order language. There are some good reasons for this distinction, despite it being patronizing—the sensitive reader can adopt the attitude that it's me, not him or her, who needs to keep from getting confused!

So below we consider what happens when both of these extra generalities are introduced. This is related to (but is done somewhat differently than in) **dynamic logic**. Certainly we'll be less general than the latter's most abstract form, which, for example, has non-deterministic constructs—the output sometimes not being unique for a given input to a terminating program.

First we remark on how this extra generality (perhaps unexpectedly) expresses much more than the F-H *partial* correctness from which we started. But the spirit here is merely intellectual interest—no claim is made about the languages below being just the right ones for all the important practical questions about programs.

Here is some notation, part of which is stolen from dynamic logic.

The language consisting of **assertions**, denoted $\mathcal{A}$, $\mathcal{B}$, etc., depending on a given $1^{\underline{\text{st}}}$order language plus its associated 'while' command language, will be as follows :

atomic assertions: $[F : C : G]$

for F, G in the underlying $1^{\underline{\text{st}}}$order language; C from its 'while' language;

general assertions: $\neg\mathcal{A}$ $(\mathcal{A}\&\mathcal{B})$ $\forall x\mathcal{A}$ $[\mathcal{A} : C : \mathcal{B}]$

The last of these is needed for simplifying the formulation of the proof system.

Now use the following abbreviations :

$(\mathcal{A} \Longrightarrow \mathcal{B})$ abbreviates the assertion $\neg(\mathcal{A}\&\neg\mathcal{B})$;

$(\mathcal{A} \Longleftrightarrow \mathcal{B})$ abbreviates the assertion $((\mathcal{A} \Longrightarrow \mathcal{B})\&(\mathcal{B} \Longrightarrow \mathcal{A}))$;

Bracket removal abbreviations will be natural ones as in **[LM]**, no memorizing precedence rules! For example, only the outside brackets should disappear on that last string. But "&" is 'stickier' than both "$\Longrightarrow$" and "$\Longleftrightarrow$", as everyone knows without needing to memorize it!

Tr (for "true") abbreviates the formula $x_0 \approx x_0$;

Nu (for "null") abbreviates the command $x_0 \div x_0$;

$[C]G$ abbreviates the assertion $[Tr : C : G]$;

'G' abbreviates the assertion $[Nu]G$, that is, $[Tr : Nu : G]$;

$[C]\mathcal{A}$ abbreviates the assertion $[`Tr` : C : \mathcal{A}]$;

$<C>G$ abbreviates the assertion $\neg[C]\neg G$;

and

$<C>\mathcal{A}$ abbreviates the assertion $\neg[C]\neg\mathcal{A}$.

These abbreviations each get their semantics from the basic semantics for $[F : C : G]$ given above, plus the usual semantics for $\neg$, $\&$ and $\forall x$. The semantics for $[\mathcal{A} : C : \mathcal{B}]$ is exactly as given at the beginning of the addendum for its dwarf cousin $[F : C : G]$. This can be re-worded to say that $[\mathcal{A} : C : \mathcal{B}]$ is true at a given $\underline{v}$ exactly when [*$\mathcal{A}$ is false at $\underline{v}$, or $||C||(\underline{v}) = err$, or $\mathcal{B}$ is true at $||C||(\underline{v})$*].

Recall also that $\forall x\mathcal{A}$ is true at $\underline{v}$ iff $\mathcal{A}$ is true at all states $\underline{w}$ which agree with $\underline{v}$ except possibly at the variable x. When $\underline{v}$ and $\underline{w}$ are related like this, we abbreviate this to $\underline{v} \ \approx_x \ \underline{w}$. This may also be written : $\underline{v}(y) = \underline{w}(y)$ for all variables y except possibly $y = x$.

Thus 'G' is true at $\underline{v}$ iff G is true at $\underline{v}$ (in the normal $1^{\underline{\text{st}}}$order logic sense). So the new language can be thought of as an extension of $1^{\underline{\text{st}}}$order logic, and many would regard maintaining the distinction between G and 'G' as unnecessary.

As above, the F-H statement $F\{C\}G$ can be taken to be any assertion

$$\forall y_1 \cdots \forall y_k \ [F : C : G]$$

where the y_i include all the variables in C and every free variable in F or G.

It is a simple exercise to show that $[F : C : G]$ and 'F' $\Longrightarrow [C]G$ have exactly the same semantics; that is, they are *truth equivalent*.

But now consider the semantics of

$$`F` \Longrightarrow <C>G \ .$$

This is true at $\underline{v}$ iff

[either 'F' is false at $\underline{v}$ or $<C>G$ is true at $\underline{v}$] iff

[either F is false at $\underline{v}$ or $\neg[C]\neg G$ is true at $\underline{v}$] iff

[either F is false at $\underline{v}$ or $[Tr : C : \neg G]$ is false at $\underline{v}$] iff

[either F is false at $\underline{v}$ or [$||C||(\underline{v}) \neq err$ and $\neg G$ is false at $||C||(\underline{v})$]] iff

[if F is true at $\underline{v}$ then [C terminates at $\underline{v}$ and G is true at $||C||(\underline{v})$]] .

So we see that 'F' $\Longrightarrow <C> G$ *being true at all* $\underline{v}$ *from an interpretation is exactly the statement asserting the* **total** *correctness of the command* C *for precondition* F *and postcondition* G !

This is the (at least to me) slightly surprising fact which more-or-less tells us that a 'complete' theory of *partial* correctness which includes propositional connectives will necessarily include a theory of *total* correctness. (For total correctness, there no standard notation such as $F\{C\}G$ or $\{F\}C\{G\}$, which are almost universally used by partial correctologists.)

Another way of viewing this, as the dynamic logicians do, is as a kind of duality between the [ ] and <> operators, analogous to the duality between $\forall$ and $\exists$, or, more to the point, the duality between "necessity" and "possibility" operators in modal logic. But recasting everything in that style, with Kripke semantics, etc., seems to be unnecessary for us. Note however that [ ] and <> play a big rôle in expressing the proof system below. And also the final rule of inference below is similar to rules that occur in the theory of total correctness, different from any that have so-far appeared here. (It seems to be even different in a minor but essential way from what appears *anywhere* else, having the advantage over others of being sound!—see later comments on this.)

The semantics of $[C]\mathcal{A}$ and $<C>\mathcal{A}$ on their own (and similarly with G replacing $\mathcal{A}$, but we might as well just take $\mathcal{A} =$ 'G') can be easily checked to be the following:

$[C]\mathcal{A}$ is true at $\underline{v}$ iff either $||C||(\underline{v}) = err$ or $\mathcal{A}$ is true at $||C||(\underline{v})$.

$<C>\mathcal{A}$ is true at $\underline{v}$ iff both $||C||(\underline{v}) \neq err$ and $\mathcal{A}$ is true at $||C||(\underline{v})$.

Now we present a proof system for this [language plus semantics]. We must back off from previous generality and assume that the underlying $1^{\underline{\text{st}}}$order language is number theory and that the interpretation is $\mathbf{N}$. (See **[Harel]**, p. 29, for a notion of *arithmetical universe* which generalizes this.) We shall continue to deal only with derivations in which the premiss set is the highly unruly set of all 'F' for which F is true in $\mathbf{N}$.

Here is the system which gives the completeness theorem below for this assertion language. The first seven are axioms, and the rest are rules with at least one premiss.

The System

$(\mathrm{I})\mathcal{A} \Longrightarrow \mathcal{A}\&\mathcal{A} \quad (\mathrm{II})\mathcal{A}\&\mathcal{B} \Longrightarrow \mathcal{A} \quad (\mathrm{III})(\mathcal{A} \Longrightarrow \mathcal{B}) \Longrightarrow (\neg(\mathcal{B}\&\mathcal{C}) \Longrightarrow \neg(\mathcal{C}\&\mathcal{A}))$

These three plus (MP), Rosser's system, can be replaced by any axioms which, with (MP), give a complete system for classical propositional logic, a so-called 'Hilbert-style proof system'. This is referred to below as "*propositional completeness*".

(IV) $\quad [F : C : G] \iff (\text{`}F\text{`} \Longrightarrow [C]G)$

(V) $\quad [x \div t]F \iff \text{`}F^{[x \to t]}\text{`}$

(VI) $\quad [(C_1; C_2)]F \iff [C_1][C_2]F$

Notice that, on the right-hand side here, we have an assertion of the form $[C_1]\mathcal{A}$, not just $[C_1]H$ for a formula H from the underlying $1^{\underline{\text{st}}}$order language. This is really why we needed the language to include the assertion construction $[\mathcal{A} : C : \mathcal{B}]$, or at least the case of it where $\mathcal{A}$ is 'Tr'. However, expressivity will say that $[C_1][C_2]F$ 'could be replaced' by $[C_1]H$ for some $1^{\underline{\text{st}}}$order formula H.

(VII) $\quad [\mathsf{ite}(H)(C_1)(C_2)]G \iff (\text{`}H\text{`} \Longrightarrow [C_1]G)\&(\text{`}\neg H\text{`} \Longrightarrow [C_2]G)$

(VIII) $\quad \dfrac{\mathcal{A}\ ,\ \mathcal{A} \Longrightarrow \mathcal{B}}{\mathcal{B}}\ , \quad$ i.e. modus ponens, or (MP)

(IX) $\quad \dfrac{\mathcal{A} \Longrightarrow \mathcal{B}}{[C]\mathcal{A} \Longrightarrow [C]\mathcal{B}}$

(X) $\quad \dfrac{\mathcal{A} \Longrightarrow \mathcal{B}}{\forall x\ \mathcal{A} \Longrightarrow \forall x\ \mathcal{B}}$

(XI) $\quad \dfrac{[F \wedge G : C : F]}{[F : \mathsf{whdo}(G)(C) : F \wedge \neg G]}$

(XII) $\quad \dfrac{\text{`}F^{[x \to 0]} \to \neg G\text{`} \quad , \quad \text{`}F^{[x \to x+1]} \to G\text{`} \quad , \quad < F^{[x \to x+1]} : C : F >}{< \exists x F : \mathsf{whdo}(G)(C) : F^{[x \to 0]} >}$

for $x \notin C \cup G$, and defining $< J : C : K > \ := \ \text{`}J\text{`} \Longrightarrow < C > K$.

Then we get a completeness theorem by adopting methods due to Harel in dynamic logic, and of course inspired by Cook :

There is a derivation for $\mathcal{A}$ if and only if $\mathcal{A}$ is true at all states from $\mathbf{N}$.

Proving this is in detail will be somewhat lengthy.

Except for brevity in the instance of proving validity of the last rule, (XII), the soundness half of the proof is a sequence of brief and straightforward verifications of validity of each axiom and rule separately. Validity for axioms means "true in $\mathbf{N}$". Validity for rules means that, when each assertion in the numerator is true in $\mathbf{N}$, then so is the denominator. One is not claiming to be true in $\mathbf{N}$ a (stronger than the rule) axiom which says "conjunction of numerator assertions $\Longrightarrow$ denominator assertion". That would amount to changing the second previous sentence by using "true at $\underline{v}$" everywhere. The deduction lemma in general form will not be deducible in this system, which is analogous to **System** from [**LM**], rather than to **System***.

A version of that last rule, (XII), is unsoundly stated in all other sources which I have seen. More specifically, the statements in [**Apt**], Rule 6, p.441, and [**Harel**] , bottom of p.37, fail to have the restriction that the variable x does not occur in G. Here is a simple counterexample to the soundness of that stronger rule used by those much-cited papers. We now drop the silly single quotes that we've been putting around each $1^{\underline{\text{st}}}$order formula .

$$\frac{(x \approx y)^{[x \mapsto 0]} \to \neg x + 1 \approx y \ , \ (x \approx y)^{[x \mapsto x+1]} \to x + 1 \approx y \ , \ < x + 1 \approx y : \text{“}y \dot{\leftarrow} y - 1\text{”} : x \approx y >}{< \exists x \ x \approx y : \mathsf{whdo}(x + 1 \approx y)(\text{“}y \dot{\leftarrow} y - 1\text{”}) : (x \approx y)^{[x \mapsto 0]} >}$$

Explanation : This is an example of rule (XII), except that x *does* occur in G, which is taken to be $x + 1 \approx y$ for a pair of distinct variables x and y. The formula F is taken to be $x \approx y$. The command C, written "$y \dot{\leftarrow} y - 1$", is intended to be any command (easy to come by) which decreases the value of y by 1 when the input is any state where y has positive value; which may do almost anything you want when y has value zero, including perhaps looping; which never affects the value of x; and indeed, when written out properly as a 'while'-command, has no occurrence of x.

It is evident from the description of C that the total correctness assertion on the right in the numerator is true in $\mathbf{N}$. The middle formula is tautological.

The truth in **N** of the formula on the left is established without any major intellectual effort. (It's not true in **Z**, of course.)

However, the total correctness statement in the denominator is certainly not true in **N**. Just take any state $\underline{v}$ in which $\underline{v}(x) = 1 = \underline{v}(y)$. The formula on the left, $\exists x\ x \approx y$, is true there (in fact, at any state, since it's logically valid. But actually $x \approx y$ itself is true there, making this also a counterexample to validity for Harel's version, as discussed in the next paragraph.) At this state $\underline{v}$, the command $\mathsf{whdo}(G)(C)$ does no passes of C at all, since $x + 1 \approx y$ is false at $\underline{v}$. And so termination does hold, with $||\mathsf{whdo}(G)(C)||(\underline{v}) = \underline{v}$. But the formula on the right in the denominator, namely $0 \approx y$, is certainly false at $\underline{v}$, so the correctness aspect fails. Thus the assertion in the denominator is false in **N**, as required. (If desired, one can certainly give an example where the whdo-command does any pre-assigned finite number of passes—but counterexamples here are necessarily to correctness, not to termination.)

In [**Harel**], p.37, the rule we've named (XII) also has the $\exists x$ in the denominator missing. That weakens it, but the above counterexample still works to prove unsoundness. For that reason, and since we have propositional completeness, Harel's system as he states it may be used to derive any assertion whatsoever, and so one cannot claim that it is inadequate. However, if one puts our restriction that $x \notin G$ into his rule, but continues to omit the "$\exists x$", I don't know how to prove that system to be adequate, if it is, in fact, adequate. He provides all the fundamental ideas for the proof of completeness we give below, but scrimps on details which would be needed to verify by means of such a proof that his system is complete.

Example: The assertion $< C > J$ is clearly true in **N**, where

$$C \text{ is } \mathsf{whdo}(0 < y)(\text{“}y \dot{\leftarrow} y - 1\text{”}) \ , \text{ and } J \text{ is } y \approx 0 \ .$$

A derivation of it (within our system) is fairly easy to come by: Taking x to be any variable not occurring in C, rule (XII) yields a derivation of

$$\exists x\ y \approx x \ \rightarrow \ \ < C > J \ ,$$

since J is $(y \approx x)^{[x \mapsto 0]}$. The rule is applicable since the corresponding assertions in the numerator of that rule, namely

$$y \approx 0 \rightarrow \neg 0 < y \ , \ \ y \approx x{+}1 \rightarrow 0 < y \ , \ \text{and} \ \ < y \approx x{+}1 : \text{“}y \dot{\leftarrow} y{-}1\text{”} : y \approx x >$$

are clearly all derivable, the first two being premisses. The last would have an elementary proof, once the command is written out in detail, involving only the first eleven axioms and rules. But now $\exists x\ y \approx x$, being logically valid, is true in $\mathbf{N}$ and so derivable, and thus (MP) immediately yields the required result.

But this argument wouldn't work using the fixed-up rule from [**Harel**], since we have to drop the "$\exists x$", so logical validity evaporates. I suspect, and will leave it to the reader to attempt to verify, that $< C > J$ is not derivable in the latter system, which would therefore be incomplete. In fact, it seems likely that $\mathcal{A} \rightarrow < C > J$ is derivable in that system only for assertions $\mathcal{A}$ for which $\mathcal{A} \rightarrow y \approx x$ is true in $\mathbf{N}$.

In the proofs below, we shall point out where use is made of the two matters discussed above re rule (XII) (certainly essential use, in the case of $x \notin G$, and probably for $\exists x$).

Here then is a proof that the rule (XII), as we state it, is sound.

With the notation from that rule, assume that the three assertions in its numerator are true in $\mathbf{N}$.

Let $\underline{v}$ be such that $\exists x\ F$ is true at $\underline{v}$. Choose some $\underline{w}$ with $\underline{v} \underset{x}{\sim} \underline{w}$, and with F itself true at $\underline{w}$. Let $n := \underline{w}(x)$. For $0 \leq i \leq n$, define $\underline{w}^{(i)} := \underline{w}^{(x \mapsto n-i)}$, which maps x to $n - i$, but any other y to $\underline{w}(y)$. Thus $\underline{w}^{(0)} = \underline{w}$; $\underline{w}^{(n)}(x) = 0_{\mathbf{N}}$; and $\underline{w}^{(i)} \underset{x}{\sim} \underline{v}$ for all i. Now we have a \tiny diagram of implications, explained below, where $\mathbf{t}$ is short for "is true at". Of course, the \tiny $\Longrightarrow$'s and $\Downarrow$'s in this diagram are from the metalanguage of this writeup, *not* from the formal language of assertions!

$$
\begin{array}{ccccccc}
F\mathbf{t}\underline{w}^{(0)} & \Rightarrow & F^{[x \to x+1]}\mathbf{t}\underline{w}^{(1)} & \Rightarrow & F\mathbf{t}||C||(\underline{w}^{(1)}) & \Rightarrow & F^{[x \to x+1]}\mathbf{t}||C||(\underline{w}^{(2)}) \\
\Downarrow & & \Downarrow & & \Downarrow & & \Downarrow \\
G\mathbf{t}\underline{w}^{(1)} & & G\mathbf{t}||C||(\underline{w}^{(2)}) & & G\mathbf{t}||C||^{n-1}(\underline{w}^{(n)}) & & F^{[x \to 0]}\mathbf{t}||C||^{n}(\underline{w}^{(n)}) \\
\Downarrow & & \Downarrow & & \Downarrow & & \Downarrow \\
G\mathbf{t}\underline{v} & & G\mathbf{t}||C||(\underline{v}) & & G\mathbf{t}||C||^{n-1}(\underline{v}) & & \neg G\mathbf{t}||C||^{n}(\underline{w}^{(n)}) \\
 & & & & & & \Downarrow \\
 & & & & & & \neg G\mathbf{t}||C||^{n}(\underline{v})
\end{array}
$$

On the top row, the 1st, 3rd, 5th, etc. $\Longrightarrow$'s are correct because, quite generally, F being true at $\underline{u}^{(x \mapsto i)}$ implies that $F^{[x \to x+1]}$ is true at $\underline{u}^{(x \mapsto i-1)}$. We need also to use that, because $x \notin C$, one has $||C||^j(\underline{w}^{(i)}) = ||C||^j(\underline{w})^{(i)}$.

On the top row, the 2nd, 4th, etc. $\Longrightarrow$'s follow because of the truth of $< F^{[x \to x+1]} : C : F >$ in $\mathbf{N}$ (the rightmost assertion inside the numerator of the rule). It gives us that, when $F^{[x \to x+1]}$ is true at $\underline{u}$, it follows that F is true at $||C||(\underline{u})$, which is $\neq err$ there. Part of the intended information in the diagram is that each state appearing there is $\neq err$, any questionable case following because of the $\Longrightarrow$ coming into the spot where that state appears.

As for the downward implications which are *not* in the rightmost column, those in the upper row are correct immediately from the truth of $F^{[x \to x+1]} \rightarrow G$ (the middle formula from the numerator of the rule). That those in the second row are correct comes from observing that $\underline{w}^{(1)} \ \curlyvee \ \underline{v}$, then that $||C||(\underline{w}^{(2)}) \ \curlyvee \ ||C||(\underline{v})$ because $\underline{w}^{(2)} \ \curlyvee \ \underline{v}$ and $x \notin C$, then that $||C||^2(\underline{w}^{(3)}) \ \curlyvee \ ||C||^2(\underline{v})$, etc. $\cdots$; and also using the quite general fact that $[\underline{u} \ \curlyvee \ \underline{z}$ and $x \notin G]$ implies that G is true at $\underline{u}$ if and only if G is true at $\underline{z}$. (This is the one place where $x \notin G$ is used. When proving adequacy, we'll point out how the restriction is not a problem.)

As for the downward implications which *are* in the rightmost column :

The top one is correct because of the general fact that if $\underline{u}(x) = 0_{\mathbf{N}}$, then H is true at $\underline{u}$ iff $H^{[x \to 0]}$ is true there, and the special fact that because $\underline{w}^{(n)}(x) = 0_{\mathbf{N}}$ and $x \notin C$, we get $||C||^i(\underline{w}^{(n)})(x) = 0_{\mathbf{N}}$ for all i .

The middle one is correct simply because $F^{[x \to 0]} \rightarrow \neg G$ (the leftmost formula from the numerator of the rule) is true in $\mathbf{N}$.

And the bottom one is correct just as with the bottom downward implications in the other columns.

Now that you buy the correctness of the diagram, we can finish quite quickly the proof that the assertion from the denominator of the rule is true in $\mathbf{N}$. The statements at the bottoms of all the columns in the diagram immediately show that the 'while-do' command does terminate on input $\underline{v}$. More precisely

$$||\mathsf{whdo}(G)(C)||(\underline{v}) \ = \ ||C||^n(\underline{v}) \ \neq \ err \ .$$

But now the formula $F^{[x \to 0]}$ is true at that state, as required. That last claim is just what appears as the second-from-top statement in the last column of the diagram, except that $\underline{v}$ has been replaced by $\underline{w}^{(n)}$. But that's no problem,

as those two states are related by $\sim_x$, and the formula in question has no free x, as it has been substituted by 0.

The three underlined statements in this proof say exactly that the assertion from the denominator of the rule is true in **N**, as required.

Proof of adequacy (completeness) of the system.

We must prove, for assertions $\mathcal{A}$:

$(*)$ if $\mathcal{A}$ is true in **N**, then $\mathcal{A}$ is derivable.

It will be convenient (and of educational value) to pass over to a slightly different language. In this language, each $1^{\underline{\text{st}}}$order formula will actually *be* an assertion (so we get rid of those silly single quotes 'F'). The main change is to use directly, not as an abbreviation, the string $[C]\mathcal{A}$, and we get rid of the $[\mathcal{A} : C : \mathcal{B}]$, which will be replaced by $A \Longrightarrow [C]\mathcal{B}$. This language will really be a specialized form of a dynamic logic language.

The language is defined in the usual structural inductive fashion :
Each atom will be an *atomic* $1^{\underline{\text{st}}}$order formula . (These are the formulas $s \approx t$ and $s < t$, for any terms s and t, built up using variables and the symbols $+$, $\times$, 0 and 1.)
Compound assertions are built as

$$\mathcal{A} \mapsto \neg\mathcal{A} \;\; ; \;\; (\mathcal{A}, \mathcal{B}) \mapsto (\mathcal{A} \wedge \mathcal{B}) \;\; ; \;\; \mathcal{A} \mapsto \forall x \; \mathcal{A} \;\; ; \;\; \text{and} \;\; \mathcal{A} \mapsto [C]\mathcal{A} \; .$$

The first three of these four build any $1^{\underline{\text{st}}}$order formula of course. We'll revert to $\neg$, $\wedge$ and $\rightarrow$, to conform with the notation used in **[LM]**, our canonical reference for $1^{\underline{\text{st}}}$order logic. (That gives the reward of letting us use $\Longrightarrow$ in the metalanguage of this discussion without fussy comments!)

The new proof system will be obtained simply from the old one :
(1) drop the single quotes on each $1^{\underline{\text{st}}}$order formula ;
(2) omit axiom (IV), which is meaningless for the new language anyway;
(3) alter rules (XI) and (XII) to

$$\frac{F \wedge G \rightarrow [C]F}{F \;\rightarrow\; [\mathsf{whdo}(G)(C)](F \wedge \neg G)}$$

and

$$\frac{F^{[x \to 0]} \rightarrow \neg G \quad , \quad F^{[x \to x+1]} \;\rightarrow\; (G \wedge < C > F)}{\exists x F \rightarrow < \mathsf{whdo}(G)(C) > F^{[x \to 0]}} \qquad \text{for } x \notin C \cup G \; .$$

In this new language, as before, define $< C > F \ := \ \neg[C]\neg F$.

The semantics of the new language is exactly the same as that of the old. But what is a definition here for, say, the primitive string $[C]F$ in the new language, was a little proposition for that abbreviation in the old language.

Now I claim that proving $(*)$ above for the new language and proof system suffices to get it for the old. That seems fairly obvious, but let us at least give a sketch of how to check this.

Use $\mathcal{L}_{\rm old}$, $\mathcal{L}_{\rm new}$, $\rm sys_{old}$, $\rm sys_{new}$ in the obvious way for the old and new languages and proof systems. The original $(*)$ says

$$\mathbf{N} \models \mathcal{O} \quad \text{implies} \quad \rm sys_{old} \vdash \mathcal{O}$$

for any $\mathcal{O}$ in the old language. Below we shall prove

$$\mathbf{N} \models \mathcal{N} \quad \text{implies} \quad \rm sys_{new} \vdash \mathcal{N}$$

for any $\mathcal{N}$ in the new language. In both cases, the $\vdash$ means to use the system, and to take every $1^{\underline{\rm st}}$order formula which is true in $\mathbf{N}$ as a premiss.

Define, by induction on structure, a map

$$\varphi \ : \ \mathcal{L}_{\rm old} \ \rightarrow \ \mathcal{L}_{\rm new} \ ,$$

via

$$\varphi([F : C : G]) \ := \ F \rightarrow [C]G \ ;$$

$$\varphi(\neg\mathcal{A}) := \neg\varphi(\mathcal{A}) \ ; \ \varphi(\mathcal{A} \wedge \mathcal{B}) := \varphi(\mathcal{A}) \wedge \varphi(\mathcal{B}) \ \ ; \ \ \varphi(\forall x \ \mathcal{A}) := \forall x \ \varphi(\mathcal{A}) \ ;$$

and

$$\varphi([\mathcal{A} : C : \mathcal{B}]) \ := \ \varphi(\mathcal{A}) \rightarrow [C]\varphi(\mathcal{B}) \ .$$

Then it is mildly tedious checking to show

$$\mathbf{N} \models \mathcal{O} \quad \text{if and only if} \quad \mathbf{N} \models \varphi(\mathcal{O}) \ ,$$

and

$$\rm sys_{old} \vdash \mathcal{O} \quad \text{if and only if} \quad sys_{new} \vdash \varphi(\mathcal{O}) \ .$$

These obviously do the needed job (in fact, just the "only if " in the first and the "if " in the second). We'll leave it to the reader to go through the checking, proceeding by induction on $\mathcal{O}$, which carefully verifies the two displays just above.

If that's disagreeable, you can think of the completeness of the new system as being the main point of this addendum. And regard the old language as just motivation being used to facilitate the passage from classical Floyd-Hoare logic to this concrete subtheory of dynamic logic.

Before proving adequacy for the new system, we need an **expressiveness result**, namely :

For any assertion $\mathcal{A}$ in the new language, there is a $1^{\underline{\text{st}}}$*order formula A such that $A \leftrightarrow \mathcal{A}$ is true in* $\mathbf{N}$.

The initial and three of the four inductive cases (of the proof by structural induction on $\mathcal{A}$) are very easy. The other case, when $\mathcal{A}$ is $[C]\mathcal{B}$, can, as with the proof of **8.3**, be done easily with the help of Gödel's definability result for semi-decidable relations, as follows :

Firstly, it's clearly equivalent (and turns out to seem linguistically simpler) to deal with $< C > \mathcal{B}$ instead of $[C]\mathcal{B}$. Let all the variables in C and the free ones in $\mathcal{B}$ be from among the distinct variables $y_1, \cdots, y_n$. Let $x_1, \cdots, x_n$ be distinct variables, disjoint from the y_i's. For the fixed command C, define a $2n$-ary relation, R_C , on natural numbers by

$$R_C(a_1, \cdots, a_n, b_1, \cdots, b_n) \iff ||C||(\vec{a}) = \vec{b} \ .$$

This is semi-decidable (indeed, the archetype of such!), so, by Gödel, let H be a $1^{\underline{\text{st}}}$order formula with free variables from among the x_i and y_i, such that

$$R_C(\vec{a}, \vec{b}) \iff H^{[\vec{x} \to \vec{a} \ , \ \vec{y} \to \vec{b}]} \text{ is true in } \mathbf{N} \ .$$

By the inductive hypothesis, choose a $1^{\underline{\text{st}}}$order formula B with free variables from among the y_i, such that $B \leftrightarrow \mathcal{B}$ is true in $\mathbf{N}$.

Now define A to be $\exists y_1 \cdots \exists y_n \ (H \wedge B)$.
Then

$$A \text{ is true at } \vec{a} \iff \text{for some } \vec{b} \ , \ (H \wedge B)^{[\vec{y} \to \vec{b}]} \text{ true at } \vec{a} \iff$$

$$\text{for some } \vec{b} \ , \text{ both } H^{[\vec{x} \to \vec{a} \ , \ \vec{y} \to \vec{b}]} \text{ and } B^{[\vec{y} \to \vec{b}]} \text{ are true in } \mathbf{N} \iff$$

$$\text{for some } \vec{b} \ , \text{ both } R_C(\vec{a}, \vec{b}) \text{ holds and } B \text{ is true at } \vec{b} \iff$$

$$\text{for some } \vec{b} \ , \ ||C||(\vec{a}) = \vec{b} \text{ and } B \text{ is true at } ||C||(\vec{a}) \iff$$

$$||C||(\vec{a}) \neq err \text{ and } \mathcal{B} \text{ is true at } ||C||(\vec{a}) \iff < C > \mathcal{B} \text{ is true at } \vec{a} \ .$$

Thus, as required, $A \leftrightarrow \ < C > \mathcal{B}$ is true in $\mathbf{N}$.

Now let's proceed to the proof of adequacy for the new system. Accuse an occurrence of $\forall x$ in $\mathcal{A}$ of being dull when its scope is a $1^{\underline{\text{st}}}$order formula. Define, for $\mathcal{A}$ in the new language,

$$M_{\mathcal{A}} \ := \ \#\text{non-dull occurrences in } \mathcal{A} \text{ of } \forall x \text{ for various } x \ + \\ \#\text{occurrences in } \mathcal{A} \text{ of } [C] \text{ for various } C \ .$$

Then $\mathcal{A}$ is a $1^{\underline{\text{st}}}$order formula if and only if $M_{\mathcal{A}} = 0$. (As it happens, if #occurrences in $\mathcal{A}$ of $[C]$ is 0, then every occurrence of $\forall x$ is in fact dull.)

We shall prove $(*)$ (for $\mathcal{A}$ from the new language) by induction on $M_{\mathcal{A}}$. Since all $1^{\underline{\text{st}}}$order formulas are premisses, the start of the induction is trivial.

Here is the overall sketch of the lengthy inductive step, in reverse order :

First reduce it to proving $(*)$ for assertions $\mathcal{A} \rightarrow \mathsf{sym}\mathcal{B}$, in the four cases of the symbol string

$$\mathsf{sym} \ = \ [C] \ \text{ or } \ \neg[C] \ \text{ or } \ \forall x \ \text{ or } \ \neg\forall x \ ,$$

where the latter two cases involve non-dull occurrences (that is, $\mathcal{B}$ is not a $1^{\underline{\text{st}}}$order formula).

Further reduce it to proving $(*)$ for assertions of the form $F \longrightarrow [C]G$ and $F \longrightarrow \neg[C]G$ for $1^{\underline{\text{st}}}$order formulas F and G. (So we are down to two particular cases of when $M_{\mathcal{A}} = 1$.)

Finally give separate, somewhat parallel, lengthy arguments for those two cases.

Now we proceed to do these in reverse order.

Proof of $(*)$ for $\mathcal{A}$ of the form $F \longrightarrow [C]G$.

This proceeds by structural induction on C, simultaneously for all $1^{\underline{\text{st}}}$order formulas F and G.

When C is $x \dot{\leftarrow} t$: By (V), $G^{[x \rightarrow t]} \ \longrightarrow \ [x \ \dot{\leftarrow} \ t]G$ is derivable, so it remains to show $F \longrightarrow G^{[x \rightarrow t]}$ is derivable, and then to apply hypothetical syllogism from propositional completeness. But the $1^{\underline{\text{st}}}$order formula $F \longrightarrow G^{[x \rightarrow t]}$ is a premiss, since it is true in $\mathbf{N}$, because both $F \longrightarrow [x \dot{\leftarrow} t]G$ [by assumption] and $[x \dot{\leftarrow} t]G \ \longrightarrow \ G^{[x \rightarrow t]}$ [soundness of half-rule (V)] are true in $\mathbf{N}$.

When C is $\mathsf{ite}(H)(C_1)(C_2)$: Use (VII) more-or-less as we used (V) just above. We need only show

$$F \ \longrightarrow \ (H \longrightarrow [C_1]G) \wedge (\neg H \longrightarrow [C_2]G)$$

to be derivable. By propositional completeness, this reduces to the derivability separately of

$$F \longrightarrow (H \longrightarrow [C_1]G)) \quad \text{and} \quad F \longrightarrow (\neg H \longrightarrow [C_2]G) \ ,$$

or indeed

$$F \wedge H \longrightarrow [C_1]G) \quad \text{and} \quad F \wedge \neg H \longrightarrow [C_2]G \ .$$

The latter two are immediate, by the induction on C, since the truth in $\mathbf{N}$ of the latter formulae is clear from the truth of $F \longrightarrow [C]G$.

When C is $(C_1; C_2)$: Here axiom (VI) is of course crucial, but expressivity (not unexpectedly!) is also involved. By expressivity, choose a 1^{st}order formula H so that $H \longleftrightarrow [C_2]G$ is true in $\mathbf{N}$. By the structural induction on C, the assertion $H \longrightarrow [C_2]G$ is derivable. So, by (IX), the assertion $[C_1]H \longrightarrow [C_1][C_2]G$ is derivable. By hypothetical syllogism, and since $[C_1][C_2]G \longrightarrow [C]G$ is derivable, [i.e. 'is' half of axiom (VI)], it remains only to show that the assertion $F \longrightarrow [C_1]H$ is derivable. This follows from the inductive hypothesis as long as that assertion is true. But in

$$F \ \longrightarrow \ [(C_1; C_2)]G \ \longrightarrow \ [C_1][C_2]G \ \longrightarrow \ [C_1]H$$

we have that 'each $\longrightarrow$' is true, respectively, by assumption, by soundness of half-(VI), and essentially by soundness of half-(IX), knowing $[C_2]G \longrightarrow H$ to be true.

When C is $\mathsf{whdo}(H)(D)$: Using expressivity, choose a 1^{st}order formula J with $J \longleftrightarrow [C]G$ true in $\mathbf{N}$.

I claim that each of the following is derivable :

$$(1)\ F \to J \ ; \quad (2)\ J \to [C](J \wedge \neg H) \ ; \quad (3)\ J \wedge \neg H \to G \ .$$

If so, then (3) and axiom (IX) give $[C](J \wedge \neg H) \to [C]G$ to be derivable. So the double application of hypothetical syllogism to that assertion, (2) and (1) yields the desired derivation of $F \to [C]G$.

Verifying (1) : Both '$\to$'s in $F \to [C]G \to J$ are true (by assumption and choice of J), and so $F \to J$ is a premiss.

Verifying (2) : The new rule (XI) :

$$\frac{J \wedge H \to [D]J}{J \to [\mathsf{whdo}(H)(D)](J \wedge \neg H)} \quad ,$$

leaves us only to derive $J \wedge H \rightarrow [D]J$. That follows from its truth in **N** by the overall induction on C of this part of the proof. To see its truth : for a contradiction, suppose $\underline{v}$ is such that $J \wedge H$ is true at $\underline{v}$, $||D||(\underline{v}) \neq err$, and J is false at $||D||(\underline{v})$. From 'while-do' semantics and the truth of H at $\underline{v}$, we get

$$||C||(\underline{v}) \;=\; ||C||(||D||(\underline{v})) \;.$$

Thus J being false at $||D||(\underline{v})$ gives, by the specification of J, that $[C]G$ is false at $||D||(\underline{v})$. Thus the right-hand side of the display is $\neq err$, and G is false there.

So the left-hand side of the display is $\neq err$. But then, since J, therefore $[C]G$, is true at $\underline{v}$, we see that G is true at the left-hand side of the display.

The underlined statements contradict the display.

Verifying (3) : That $1^{\underline{\text{st}}}$order formula is a premiss, since its truth in **N** is easy to see : If $J \wedge \neg H$ is true at $\underline{v}$, then so are :

$\neg H$, giving $||C||(\underline{v}) = \underline{v}$, and

J , and so $[C]G$, giving either $||C||(\underline{v}) = err$ or G true at $||C||(\underline{v})$.

So, indeed, G is true at $\underline{v}$, as required.

Proof of $(*)$ for $\mathcal{A}$ of the form $F \longrightarrow \neg[C]G$.

Since $< C > G := \neg[C]\neg G$, derivability of $F \longrightarrow \neg[C]G_1$ (for all F and G_1 where it's true) is equivalent to derivability of $F \longrightarrow < C > G$ (for all F and G where *it* is true)—by 'taking G and G_1 to be negations of each other', so to speak. Proving the latter also proceeds by structural induction on C, simultaneously for all F and G, quite analogously to the previous proof, though the last case, of a 'while-do' command, is somewhat harder.

Indeed, the first three cases may be done almost word-for-word as before, changing [ ] to <> everywhere. For this, we need only show to be derivable the assertions $(\text{V})_{<>}$, $(\text{VI})_{<>}$, and $(\text{VII})_{<>}$, these being the corresponding axioms with [ ] changed to <> everywhere. To check them :

$$< x \div t > F \;=\; \neg[x \div t]\neg F \;\longleftrightarrow\; \neg(\neg F)^{[x \rightarrow t]} \;=\; \neg\neg F^{[x \rightarrow t]} \;\longleftrightarrow\; F^{[x \rightarrow t]} \;.$$

The first '$\longleftrightarrow$' is derivable using (V) and propositionality; and the second is just propositionality. For (VI), write down

$$< C_1 >< C_2 > F \;=\; \neg[C_1]\neg\neg[C_2]\neg F \;\longleftrightarrow\; \neg[C_1][C_2]\neg F$$
$$\longleftrightarrow\; \neg[(C_1;C_2)]\neg F \;=\; < (C_1;C_2) > F \;,$$

and say a few words. The last one even gives some minor propositional subtlety :

$$< \mathsf{ite}(H)(C_1)(C_2) > G \;=\; \neg[\mathsf{ite}(H)(C_1)(C_2)]\neg G$$
$$\longleftrightarrow\; \neg((H \longrightarrow [C_1]\neg G) \wedge (\neg H \longrightarrow [C_2]\neg G))$$
$$\longleftrightarrow\; \neg((H \longrightarrow \neg < C_1 > G) \wedge (\neg H \longrightarrow \neg < C_2 > G))$$
$$\longleftrightarrow\; \neg((< C_1 > G \longrightarrow \neg H) \wedge (< C_2 > G \longrightarrow H))$$
$$\longleftrightarrow\; (H \longrightarrow < C_1 > G) \wedge (\neg H \longrightarrow < C_2 > G) \;.$$

Each '$\longleftrightarrow$' is in fact derivable in the system, the last one being the mildly subtle fact that

$$\neg((J \longrightarrow \neg H) \wedge (K \longrightarrow H)) \qquad \text{and} \qquad (H \longrightarrow J) \wedge (\neg H \longrightarrow K)$$

are derivable from each other in propositional logic.

This leaves only one case in the structural inductive proof of the derivability of $F \rightarrow < C > G$ when that assertion is true, namely the case

When C is $\mathsf{whdo}(H)(D)$:
Pick a $1^{\underline{\text{st}}}$order formula J and a variable $x \notin D \cup G \cup H$ such that
J is true at $\underline{v}$ if and only if : with $n := \underline{v}(x)$ we have

(α) $||D||^n(\underline{v}) \neq err$ and H is true at $||D||^i(\underline{v})$ for $0 \leq i < n$;
and
(β) $G \wedge \neg H$ is true at $||D||^n(\underline{v})$.

To prove that J can be found, first define a command

$$D^{\#} \;:=\; \mathsf{whdo}(0 < x)(D \;;\text{“}x \Leftarrow x - 1\text{”}) \;.$$

Let $\vec{y} = (y_1, y_2, \cdots, y_r)$ be a list of distinct variables including all those in $D \cup G \cup H$. Here, as earlier, the command “$x \Leftarrow x - 1$” has the effect, when the value of x is positive, of decreasing it by 1 without altering any y_i (but it can behave arbitrarily otherwise.) We'll 'contract' $\underline{v}$ down to just writing the relevant variables. Then

$$||D^{\#}||(n, \vec{a}) \;=\; (0, ||D||^n(\vec{a})) \;,$$

where the first coordinate is the value of x and the vector coordinate is the value of $\vec{y}$ [and $(0, err)$ on the right-hand side would mean just err of course].

Now use expressivity for the assertion $<D^{\#}> H$ to find a $1^{\underline{\text{st}}}$order formula K such that

$$K^{[x\mapsto i\ ,\ \vec{y}\mapsto\vec{a}]} \text{ is true in } \mathbf{N} \iff ||D^{\#}||(i,\vec{a}) \neq err \text{ and } H \text{ is true there .}$$

Use expressivity again, this time for the assertion $<D^{\#}> (G \wedge \neg H)$ to find a $1^{\underline{\text{st}}}$order formula L such that

$$L^{[x\mapsto n\ ,\ \vec{y}\mapsto\vec{a}]} \text{ is true in } \mathbf{N} \iff ||D^{\#}||(n,\vec{a}) \neq err \text{ and } (G\wedge\neg H) \text{ is true there .}$$

Then let z be a 'new' variable, and define the required formula by

$$J \ := \ \forall z\ ((z < x \to K^{[x\mapsto z]}) \wedge (z \approx x \to L^{[x\mapsto z]}))\ .$$

Then

$$J \text{ is true at } (n,\vec{a}) \iff J^{[x\mapsto n\ ,\ \vec{y}\mapsto\vec{a}]} \text{ is true in } \mathbf{N} \iff$$

$$\forall z\ ((z < n \to K^{[x\mapsto z\ ,\ \vec{y}\mapsto\vec{a}]}) \wedge (z \approx n \to L^{[x\mapsto z\ ,\ \vec{y}\mapsto\vec{a}]})) \text{ is true in } \mathbf{N} \iff$$

$$\text{true in } \mathbf{N} \text{ are : } K^{[x\mapsto i\ ,\ \vec{y}\mapsto\vec{a}]} \text{ for } 0 \leq i < n, \text{ and } L^{[x\mapsto n\ ,\ \vec{y}\mapsto\vec{a}]} \iff$$

$$||D^{\#}||(i,\vec{a}) \neq err \text{ and } H \text{ is true there for } 0 \leq i < n \text{ and}$$

$$||D^{\#}||(n,\vec{a}) \neq err \text{ and } G \wedge \neg H \text{ is true there} \iff$$

$$||D||^n(\vec{a}) \neq err, \text{ the formula } G\wedge\neg H \text{ is true there, and } H \text{ is true at } ||D||^i(\vec{a}) \text{ for } 0 \leq i < n\ ,$$

as required.

I claim that the following are all true in $\mathbf{N}$.

$$(1)\ J^{[x\mapsto 0]} \to G \wedge \neg H\ ; \quad (2)\ J^{[x\mapsto x+1]} \to H \wedge < D > J\ ; \quad (3)\ F \to \exists x\ J\ .$$

Suspending disbelief in this for the moment, the following are then derivable :

$$J^{[x\mapsto 0]} \to \neg H \quad ; \quad J^{[x\mapsto x+1]} \to H \quad ; \quad J^{[x\mapsto x+1]} \to < D > J\ ;$$

the first two being premisses, and the latter because it is true and therefore derivable by the induction on C.

By rule (XII) and since $x \notin D \cup H$, we get a derivation of

$$\exists x\ J \ \to \ < C > J^{[x\mapsto 0]}\ .$$

But the $1^{\underline{\text{st}}}$order formula $J^{[x\mapsto 0]} \to G$ is a premiss, and therefore the assertion $< C > J^{[x\mapsto 0]} \to < C > G$ is derivable by a rule which is (IX) except with $[C]$ replaced by $< C >$, and which is easily verified from (IX) and the definition of $< C > F$. Finally $F \to \exists x\ J$ is true, so derivable. A double dose of hypothetical syllogism then shows $F \to < C > G$ to be derivable, as required.

It remains to check the truth of (1), (2) and (3), using the assumed truth of $F \to < \mathsf{whdo}(H)(D) > G$.

(1) Assume $J^{[x\mapsto 0]}$ is true at $\underline{w}$. Then J itself is true at $\underline{v} := \underline{w}^{(x\mapsto 0)}$. So $n = 0$ in the specification defining J. From (β) we get that $G \wedge \neg H$ is true at $\underline{v}$. But since $\underline{v} \ \overset{x}{\sim}\ \underline{w}$ and $x \notin G \cup H$, we get, as required, that $G \wedge \neg H$ is true at $\underline{w}$.

(2) Assume that $J^{[x\mapsto x+1]}$ is true at $\underline{w}$.

Then J itself is true at the state $\underline{v} := \underline{w}^{(x\mapsto \underline{w}(x)+1)}$. In the specification of J, we have $n = \underline{v}(x) = \underline{w}(x) + 1 > 0$. So $i = 0$ occurs in $(\alpha)_{\underline{v},n}$, and we get that H is true at $||D||^0(\underline{v}) = \underline{v}$. So H is true also at $\underline{w}$ (which is half the required), since $\underline{v} \ \overset{x}{\sim}\ \underline{w}$ and $x \notin H$.

But, since $n \geq 1$, condition $(\alpha)_{\underline{v},n}$ also gives $||D||(\underline{v}) \neq err$. And so $||D||(\underline{w}) \neq err$, since $\underline{v} \ \overset{x}{\sim}\ \underline{w}$ and $x \notin D$. Furthermore, J is true at $||D||(\underline{w})$: We check this using the definition of J as follows. Here

$$||D||(\underline{w})(x) = \underline{w}(x) = n - 1 \ ,$$

so we must verify conditions $(\alpha)_{||D||(\underline{w}),n-1}$ and $(\beta)_{||D||(\underline{w}),n-1}$.

For the first, note that $||D||^{n-1}(||D||(\underline{w})) = ||D||^n(\underline{w}) \neq err$, since we have $||D||^n(\underline{v}) \neq err$. Also, for $0 \leq i < n - 1$, the formula H is true at $||D||^i||D||(\underline{w})) = ||D||^{i+1}(\underline{w})$, since again, $\underline{w}$ may be replaced by $\underline{v}$, and because $i + 1 < n$, making use of $(\alpha)_{\underline{v},n}$.

For $(\beta)_{||D||(\underline{w}),n-1}$, we again use the fact that $||D||^{n-1}||D||(\underline{w})) = ||D||^n(\underline{w})$, and $G \wedge \neg H$ is indeed true there, because it is true at $||D||^n(\underline{v})$.

The underlined above show that $J^{[x\mapsto x+1]} \to < D > J$ is true in $\mathbf{N}$, completing the discussion of (2).

(3) Assume that F is true at $\underline{v}$. Our basic assumption that the assertion $F \to < \mathsf{whdo}(H)(D) > G$ is true in $\mathbf{N}$ yields that

$$||\mathsf{whdo}(H)(D)||(\underline{v}) \ = \ ||C||(\underline{v}) \ \neq \ err \ ,$$

and that G is true at $||C||(\underline{v})$. Thus, there is a (unique of course) $n \geq 0$ such that $(\alpha)_{\underline{v},n}$ and $(\beta)_{\underline{v},n}$ hold, by the semantics of 'whdo'. Now define $\underline{w}$ by $\underline{w} \ \aleph \ \underline{v}$ and $\underline{w}(x) = n$. By the definition of J, the latter formula is true at $\underline{w}$. Since $\underline{w} \ \aleph \ \underline{v}$, it is immediate from the basic (Tarski) definition of truth for an existential $1^{\underline{\text{st}}}$order formula that $\exists x \ J$ is true at $\underline{v}$, as required. (But not necessarily J itself, so more is needed to establish adequacy with a weakening of Harel's rule to make it sound, perhaps more than more!)

This finally completes the (structural induction on C) proof of $(*)$ for $\mathcal{A}$ of the form $F \longrightarrow < C > G$, and so for $\mathcal{A}$ of the form $F \longrightarrow \neg[C]G$.

To complete the (induction on $M_{\mathcal{A}}$) proof of $(*)$ for *all* $\mathcal{A}$ in the new language, let sym denote one of the symbol strings

$$[C] \quad \text{or} \quad \neg[C] \quad \text{or} \quad \forall x \quad \text{or} \quad \neg\forall x \quad .$$

Lemma. *For any $\mathcal{C}$ in the new language, there is a propositional derivation of an assertion of the form*

$$\mathcal{C} \iff \wedge_\alpha \mathcal{D}_\alpha \quad ,$$

where the finite conjunction on the right-hand side has each $\mathcal{D}_\alpha$ as a finite disjunction of assertions, each of which has the following form : either each is a $1^{\underline{\text{st}}}$order formula , or else at least one of them has the form sym$\mathcal{B}$ *for our four possible* sym*'s, where $\mathcal{B}$ is not a $1^{\underline{\text{st}}}$order formula when* sym *is either $\forall x$ or $\neg\forall x$.*

The proof of this proceeds by structural induction on $\mathcal{C}$, and is simplified by simultaneously proving the dual fact, where the prefixes "con" and "dis" trade places. The initial case is trivial. In two inductive cases, namely $\mathcal{A} \mapsto \forall x \ \mathcal{A}$; and $\mathcal{A} \mapsto [C]\mathcal{A}$, one simply has a single conjunct (resp. disjunct), which itself is a single disjunct (resp. conjunct). The inductive step for negations is easy because of proving the duals simultaneously : up to cancellation of double negations, deMorganization converts each expression from the right-hand side of the lemma to its dual. The symmetry breaks for the final case of conjuncting two assertions. For the actual lemma statement, the result is numbingly obvious. For the dual statement, it is run-of-the-mill obvious by using distributivity of $\wedge$ over $\vee$. (All we are really talking about here is conjunctive and disjunctive form.)

Since a conjunction is derivable (resp. true in $\mathbf{N}$) if and only if each conjunct has the same property (using propositionality of our system for the derivability), and since every true $1^{\underline{\text{st}}}$order formula is a premiss, the lemma reduces the question to proving $(*)$ only for assertions of the form $\mathcal{E} \vee \mathsf{sym}\mathcal{B}$. (One may use $\mathcal{E} = \ 0 \approx 1$ for any conjunct consisting of a single disjunct $\mathsf{sym}\mathcal{B}$.) Taking $\mathcal{A}$ as $\neg\mathcal{E}$,

we need only prove $(*)$ for assertions of the form $\mathcal{A} \to \mathsf{sym}\mathcal{B}$,

where, since $\mathcal{B}$ is not a $1^{\underline{\text{st}}}$order formula in two of four relevant cases, the inductive assumption applies to all assertions whose 'M-function' does not exceed that of one or the other of $\mathcal{A}$ or $\mathcal{B}$. All we are saying here is that

$$M_{\mathcal{A}\to\mathsf{sym}\mathcal{B}} \ = \ M_{\mathcal{A}} + M_{\mathcal{B}} + 1 \ .$$

By expressivity, choose A and B, each a $1^{\underline{\text{st}}}$order formula , such that both $A \longleftrightarrow \mathcal{A}$ and $B \longleftrightarrow \mathcal{B}$ are true in $\mathbf{N}$.

The proof will now be completed by considering each of the four possibilities for sym.

$\underline{\mathsf{sym} \text{ is } [C]}$: Use hypothetical syllogism after establishing the derivability of the three '$\to$' below :

$$\mathcal{A} \ \to \ A \ \to \ [C]B \ \to \ [C]\mathcal{B} \ .$$

$\mathcal{A} \to A$ is true, therefore derivable by the induction on the number M.
$B \to \mathcal{B}$ is true, therefore derivable by induction, and then so is the assertion $[C]B \to [C]\mathcal{B}$ derivable, using axiom (IX).
Derivability of $A \to [C]B$ follows from its truth and the first of the earlier agonizingly established $(*)$'s. Its truth is clear, since it 'factors into true assertions' :

$$A \ \to \ \mathcal{A} \ \to \ [C]\mathcal{B} \ \to \ [C]B \ .$$

$\underline{\mathsf{sym} \text{ is } \forall x}$: Just replace $[C]$ by $\forall x$ everywhere in the previous paragraph, and appeal to (X) rather than (IX), and get derivability of $A \to \forall x \ B$ much more easily, as a premiss.

$\underline{\mathsf{sym} \text{ is } \neg[C]}$: Just replace $[C]$ by $\neg[C]$ everywhere in the second previous paragraph, and make a few tiny wording changes :

Use $\mathcal{B} \rightarrow B$ true, so derivable by induction on the number M, to get $[C]\mathcal{B} \rightarrow [C]B$, and thence $\neg[C]B \rightarrow \neg[C]\mathcal{B}$ both derivable.

This case of course uses also the even more agonizingly established version of $(*)$ giving the derivability of $A \rightarrow \neg[C]B$.

sym is $\neg\forall x$: Write out the previous paragraph properly, then replace each $\neg[C]$ by $\neg\forall x$, again appealing simply to the 'premissiveness' of $A \rightarrow \neg\forall x\ B$, rather than needing the prickly proof of the derivability of $A \rightarrow \neg[C]\ B$.

So we're done : the proof system is indeed complete.

An exercise is to derive directly, from this system, all the rules from the first subsection of this chapter, as well as the rules at the beginning of the second subsection.

I don't yet know how to generalize this completeness to *arbitrary* underlying $1^{\underline{\text{st}}}$order languages and interpretations, nor to a system which merely extends the collection of F-H statements by adding in the propositional connectives—but these seem natural questions to consider. Nor do I understand why some denotational semanticizers require that one should use intuitionistic propositional logic here, rather than the classical logic as exemplified by modus ponens plus the first three axioms above.

Exercise. Going back to the old language beginning this addendum, show that $\neg F\{C\}\neg G$ is true in an interpretation iff there is at least one state $\underline{v}$ for which F is true at $\underline{v}$, and $||C||(\underline{v}) \neq err$, and G is true at $||C||(\underline{v})$.

It seems that one cannot get total correctness from partial correctness plus propositional connectives directly. One apparently needs one of the (interdefinable and slightly more refined) syntactic 'concepts' $[F : C : G]$ or $[C]G$ or $<C>G$ or $< F : C : G >$.

Addendum 3 : F-H Proof Systems where Recursive Programming Occurs.

Stephen Cook in [**C**] has at least the following fundamental accomplishments in this subject :

(1) isolating a notion of relative completeness which is *meaningful*, and generally (but not perhaps universally) regarded as significant;

(2) inventing the fundamental idea of the post relation (or *strongest post-condition*, as it is more popularly named) and its $1^{\underline{st}}$order definability (i.e. expressiveness), and using it to establish a benchmark completeness theorem for the **ATEN**-language (or, more accurately, the 'while-language') as in the first two subsections above;

(3) pretty much establishing the completeness and soundness of a proof system for the F-H statements concerning a command language with procedures which have parameters and global variables, as exposited at length in our third subsection.

One major sense in which this completeness work is 'incomplete' is that recursive procedures are forbidden. Here we give a command language which includes recursive programming, but vastly simplify by having no parameters. This is also simplified by avoiding declarations of variables. The semantics is given in the same style as the third section. Then we write down a proof system which is sound and complete, but don't include the proofs of the latter claims. At the end are some comments on what appear to this author to be the most (perhaps the only) reliable write-ups on systems for languages which get closer to 'real-live Algol', by having recursive declared procedures with parameters, and other features (but not so many as to contradict Clarke's famous theorem from the fourth subsection above).

The language, (operational) semantics and proof system below are closely related to those in Ch.5 of [**deB**]. He goes further in later chapters, to have both recursive programs and parameters, but avoids having nested procedures, even in Ch.5. We allow them, as only later chapters of [**deB**] apparently need to avoid nesting, or indeed really need to use denotational semantics, on which many words are devoted there. See also the comments near the end of this addendum. All the details for the soundness and adequacy of the system here have been checked, as we did in our third section, but it's quite lengthy, when done in enough detail to inspire confidence. This paper is getting far too verbose, so we'll not include the details here, risking

a crisis of confidence, not a novelty in this subject. And it is a risk—not actually writing down for public consumption such a proof seems almost inevitably to invite errors. At least there's no chance anyone would write real software using the language below (say, for landing one of those new Airbus planes in the fog), then verify the program using the proof system below. Caveat consumptor or something, as they say.

Actually, if we had followed Harel's wise comments quoted just below, this is where all the details would have been included, rather than in the tortuous 3rd subsection above!

> There is seemingly a drawback to our treatment in the fact that we do not provide tools for including any kinds of parameters in the programming language. The reason is our wanting to achieve a clarification of the mechanisms for reasoning about *pure* recursion. Our experience in digesting the literature on this subject indicates that in most of the cases the presentation of basic principles suffers from being obscured by rules for dealing with the parameters.
>
> David Harel [**HAREL**], p. 44

The language RTEN.

As before, the set of procedure identifiers, disjoint from the variables, will have members with names like p, q, etc. And, quite similar to the beginning of Section 3 above, the set $DEC \cup COM$ of declarations and commands is defined by mutual structural induction as follows :

$$D, D_1, \cdots, D_k \in DEC \ := \ \{ \ - \ || \ [p : C] \ \}$$

$$C, C_1, \cdots, C_n \in COM \ := \ \{ \ x \leftleftarrows t \mid \mathsf{call}\ p \ || \ \mathsf{ite}(H)(C_1)(C_2) \mid$$
$$\mathsf{begin}\ D_1; \cdots; D_k \ ; \ C_1; \cdots; C_n \ \mathsf{end} \ \}$$

Intuitively, the declaration $[p : C]$ is just saying "the procedure to be associated with the identifier p is to be C ". In the block-command just above, if D_i is $[p_i : K_i]$, we require the identifiers $p_1, \cdots, p_k$ to be distinct. As before, the case $k = 0 = n$ gives the 'skip' or 'null' or 'do-nothing' command, namely begin end, which really is a third atomic command. We have resurrected the **ite**-command (if-then-else) from **BTEN**, so H above is any quantifier-free formula from the underlying fixed $1^{\underline{\text{st}}}$order language $\mathcal{L}$ (with equality). But we have omitted the **whdo**-command, to shorten the treatment, with justification as follows :

Exercise. Using the semantics defined just below, show that the meaning of (i.e. the SSQ-sequence of)

$$\mathsf{begin}\ [p : \mathsf{ite}(H)(C; \mathsf{call}\ p)(\mathsf{begin\ end})]\ ;\ \mathsf{call}\ p\ \mathsf{end}$$

(a block-command with $k = 1 = n$) is the same as the SSQ-sequence of $\mathsf{whdo}(H)(C)$, as given in Section 3.

After reading in **[LM]**, Sect.IV-11: McSelfish Calculations how McCarthy's McSELF-language gives a completely general definition of *computability* without needing whdo, this exercise should come as no surprise. The command in the display above is a typical example of recursive programming, at least for the case where only *one* procedure identifier is involved. Mutual recursion with several declared procedures can also happen in interesting ways, of course.

Semantics of RTEN.

Here the extra semantics components s and δ are the same as in the third section; whereas π will be much simpler :

$$PRIDE\ \supset\ \text{finite set}\ \xrightarrow{\pi}\ COM\ \ .$$

We shall be writing strings C/π , where π is now more of a syntactic component, which could be identified with a string

$$[p_1 : K_1]\ [p_2 : K_2]\ \cdots [p_\ell : K_\ell]\ ,$$

where

$$\mathrm{domain}(\pi)\ =\ \{p_1 \prec p_2 \prec \cdots \prec p_\ell\}$$

with respect to some fixed linear order "$\prec$" on $PRIDE$, and where $\pi(p_i) = K_i$ for $1 \leq i \leq \ell$.

When it is defined, the semantic object $SSQ(C/\pi, s|\delta)$ will be a (possibly infinite) sequence of states s—no need to include the "$|\delta$-half", as in Subsection 3 above). The definition will have six cases : the three types of atomic commands, the ite-command, and the two cases ($k = 0 < n$) and $k > 0,\ n \geq 0$) of the block-command.

$C = x \div t$ or $\mathsf{begin\ end}$ or $\mathsf{begin}\ C_*\ \mathsf{end.}$ Defined the same as in Subsection 3, dropping all the $|\delta$'s.

$\underline{C = \mathsf{call}\ p.}$ The sequence $SSQ(\mathsf{call}\ p/\pi, s|\delta)$ is undefined unless p is in domain(π), in which case, with $\pi(p) = K$, it is

$$\prec\ s\ ,\ SSQ(K/\pi, s|\delta)\ \succ$$

(which might also be undefined, of course).

Remarks. Note that, just as in the third section, the definition, which we are halfway through giving, proceeds to define the nth term, SSQ_n, of the sequence by induction on n, and, for fixed n, by structural induction on C. The reader might prefer to re-write it in the stricter style we adopted at first in Section 3. Quite baffling to me are the sentence in **[deB]** on p. 150 :

> Note that, strictly speaking, the definition of $Comp(< E \mid P >)$ is not fully rigorous $\cdots$ it $\cdots$ may be made more precise (cf. the remark following Theorem A.27 of the appendix $\cdots$

and that remark :

> The non-trivial step in this approach to the definition of *Comp* is the use of the recursion theorem $\cdots$

Note that (essentially) "*Comp*" is our SSQ, "E" is π, "P" is C, and "the recursion theorem" is one of Kleene's famous results from the theory of recursive functions (see **[CM]**, IV-7.2). Although we haven't bothered with so-called 'array variables', AKA 'subscripted variables', which are irrelevant here, by not forbidding nested procedures, we have a more general command language here. But the need for machinery from recursive function theory to define operational semantics seems out of the question. In any case, a definition is either rigorous or not; being "*strictly speaking*" not "*fully* rigorous" is an odd turn of phrase!

So let us complete the definition of SSQ.

$\underline{C = \text{an ite-command.}}$ Define

$$SSQ(\mathsf{ite}(H)(C_1)(C_2)/\pi, s|\delta) := \begin{cases} \text{undefined} & \text{if free}(H) \not\subset \text{dom}(\delta); \\ \prec s, SSQ(C_1/\pi, s|\delta) \succ & \text{if } H \text{ is true at } s \circ \delta; \\ \prec s, SSQ(C_2/\pi, s|\delta) \succ & \text{if } H \text{ is false at } s \circ \delta. \end{cases}$$

$\underline{C = \mathsf{begin}\ [p : K]; D_*; C_*\ \mathsf{end}\ .}$ Define

$$SSQ(C/\pi, s|\delta) \ := \ \prec\ s\ ,\ SSQ(\mathsf{begin}\ D_*; C_*\ \mathsf{end}/\pi', s|\delta)\ \succ\ ,$$

where π' agrees with π, except that $\pi'(p) = K$, whatever the status of p with respect to π.

The assertion language and proof system.

We shall adopt the approach of the previous addendum. But here, as in Section 3 as opposed to Sections 1 and 2 and the last addendum, the basic F-H statement has the form $F\{C/\pi\}G$. That is, it includes π, as well as formulas F and G in $\mathcal{L}$ and a command C from the language **RTEN**.

So this language of assertions will start with atomic strings $[F : C/\pi : G]$ for F, G in the underlying $1^{\underline{\text{st}}}$order language; C from **RTEN**. Then it builds up compound assertions using

$$\neg\mathcal{A}\ \ ,\ \ (\mathcal{A}\&\mathcal{B})\ \ ,\ \ \forall x\mathcal{A}\ \text{ and }\ [\mathcal{A} : C/\pi : \mathcal{B}].$$

As before, the F-H statement $F\{C/\pi\}G$ is the (universal) closure of the assertion $[F : C/\pi : G]$.

We are using $\mathcal{A}$, $\mathcal{A}'$, $\mathcal{A}_1$, $\mathcal{B}$ etc. as names for assertions. And let $< F >$ be an abbreviation for $0 \approx 0\{\mathbf{begin\ end}/\pi\}F$. Alternatively, $< F >$ can be 'F_1', where F_1 is any (universal) closure for F.

The proof system

IN THE EARLIER VERSION ON THIS WEB PAGE HAD SEVERAL DRAWBACKS which I realizedafter a lot of work on a dynamic logic version of a pure recursion language. So the reader should go to numbers 14, 15 qnd 16 on this web page for a great deal of detail on this, and should wait breathlessly while I try to improve what had been here, based on that experience.

Besides the proofs of soundness and adequacy, we have also left the reader with the preliminary job of giving careful definitions of the various sets of free variables, then of the substitution notation, both for variables and for procedure identifiers. As indicated via the expressiveness hypothesis in the theorem, one will also need to use a $1^{\underline{\text{st}}}$order definability result via Gödel numbering for the 'post relation/strongest postcondition' for **RTEN**, or alternatively, a 'weakest precondition' version.

The papers of Olderog **[Old1], [Old2]** give what seems to be definitive positive results regarding complete F-H proof systems for imperative languages with procedures, allowing nesting, recursion, parameters, free variables and so-called sharing. Note that he doesn't bother with a full propositional language of F-H assertions, but most of his rules begin $\mathcal{H} \Longrightarrow \cdots$ on both top and bottom. Going to a full propositional presentation, writing some rules as axioms, and using the propositional tautology

$$(\mathcal{A} \Longrightarrow \mathcal{B}) \implies [(\mathcal{H} \Longrightarrow \mathcal{A}) \Longrightarrow (\mathcal{H} \Longrightarrow \mathcal{B})] \ ,$$

these would simplify somewhat, but it's all a matter of taste.

Trakhtenbrot, Halpern and Meyer in **[Clarke-Kozen eds]** have some disagreement with the use of 'copy rules' in the above papers (rather than denotational semantics) for defining the semantics of the language. However I haven't succeeded in finding a detailed treatment of their approach.

References and Self-References

[**Al**] Allison, Lloyd *A Practical introduction to denotational semantics.* Cambridge U. Press, Cambridge, 1989.

[**Apt**] Apt, K.R., *Ten Years of Hoare's Logic: A Survey—Part 1.* ACM Trans. Prog. Lang. Syst. 3(4), Oct. 1981, 431-483.

[**Ba**] Barendregt, H. P. *The Lambda Calculus : its syntax and semantics.* North-Holland, Amsterdam, 1984.

[**BKK-ed**] Barwise, J., Keisler, H.J. and Kunen, K. *The Kleene Symposium.* North-Holland, Amsterdam, 1980.

[**Bö-ed**] Böhm, C. *λ-calculus and computer science theory : Proceedings.* LNCS # 37, Springer-Verlag, Berlin, 1975.

[**Br-ed**] Braffort, Paul. *Computer Programming and Formal Systems.* North-Holland, Amsterdam, 1963.

[**deB**] deBakker, J.W., *Mathematical Theory of Program Correctness.* Prentice Hall, N.J., 1980.

[**B&J**] Boolos, G. S. and Jeffrey, R. C. *Computability and Logic.* (3rd ed.) Cambridge U. Press, Cambridge, 1989.

[**C**] Cook, Stephen A., *Soundness and Completeness of an Axiom System for Program Verification.* SIAM. J. COMPUT. (7), 1978, 70-90.

[**C+**] Cook, Stephen A., *Corrigendum : etc.* SIAM. J. COMPUT. 10(3), 1981, 612.

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

[**Cl1**] Clarke, E.M. Jr., *Programming Language Constructs for which it is Impossible to Obtain Good Hoare-like Axioms.* J. ACM 26, 1979, 129-147.

[**Clarke-Kozen eds**] *Logics of Programs.* Lecture Notes in CS # 164, Springer, 1984.

[**CM**] Hoffman, P. *Computability for the Mathematical.* this website, 2005.

[**Cu**] Curry, H. B., Hindley, J. R. and Seldin, J. P. *Combinatory Logic, Vol. II.* North-Holland, Amsterdam, 1972.

[**D-S-W**] Davis, M. D., Sigal, R. and Weyuker, E. J. *Computability, Complexity, and Languages.* (2nd ed.) Academic Press, San Diego, 1994.

[**En**] Engeler, Erwin, et al. *The Combinatory Programme.* Birkhäuser, Basel, 1995.

[**F**] Floyd, Robert W. and Beigle, Richard *The Language of Machines: An introduction to computability and formal languages.* Computer Science Press, New York, 1994.

[**Fi**] Fitch, F. B. *Elements of Combinatory Logic.* Yale U. Press, New Haven, 1974.

[**G**] Gordon, Michael J.C. *Programming Language Theory and its Implementation.* Prentice Hall, London, 1988.

[**Go**] Gordon, M. J. C. *Denotational Description of Programming Languages.* Springer-Verlag, Berlin, 1979.

[**Harel**] Harel, David *First-Order Dynamic Logic.* Lecture Notes in CS #68, Springer, 1979.

[**Hoare-Shepherdson eds**] *Mathematical Logic and Programming Languages.* Prentice Hall, N.J., 1985.

[**H**] Hodel, R. E. *An Introduction to Mathematical Logic.* PWS Publishing, Boston, 1995.

[**Hi**] Hindley, J. R. *Standard and Normal Reductions.* Trans.AMS, 1978.

[**HS**] Hindley, J. R. and Seldin, J. P. *Introduction to Combinators and the λ-calculus.* London Mathematical Society Student Texts # 1, Cambridge U. Press, Cambridge, 1986.

[**HS-ed**] Hindley, J. R. and Seldin, J. P. *To H. B. Curry : Essays on Combinatory Logic, lambda-calculus, and formalism.* Academic Press, London, 1980.

[**J**] Jones, N. D. *Computability and Complexity—From a programming perspective.* MIT Press, Cambridge, Mass., 1997.

[**KMA**] Kfoury, A. J., Moll, R. N. and Arbib, M. A. *A Programming Approach to Computability.* Springer-Verlag, Berlin, Heidelberg, 1982.

[**Kozen-ed.**] *Logics of Programs.* Lecture Notes in CS # 131, Springer, 1982.

[**Kl**] Klop, J. W. *Combinatory Reduction Systems.* Mathematisch Centrum, Amsterdam, 1980.

[**Ko**] Koymans, C. P. J. *Models of the lambda calculus.* CWI Tract, Amsterdam, 1984.

[**L**] Lalement, R. *Computation as logic.* Masson, Paris, 1993.

[**La-ed**] Lawvere, F. W. *Toposes, Algebraic Geometry and Logic.* LNM # 274, Springer-Verlag, Berlin, 1972.

[**LM**] Hoffman, P. *Logic for the Mathematical.* this website, 2003.

[**LS**] Lambek, J. and Scott, P. J. *Introduction to higher order categorical logic.* Cambridge U. Press, Cambridge, 1986.

[**M**] Mendelson, E. *Introduction to Mathematical Logic.* (3rd ed.) Wadsworth and Brooks/Cole, Monterey, 1987.

[**McC**] McCarthy, J. *A Basis for a Mathematical Theory of Computation,* in *Computer Programming and Formal Systems*, edited by P. Braffort and D. Hirschberg, North-Holland, Amsterdam, 1963 .

[**MeMeMe**] Webb, J. C. *Mechanism, Mentalism and Metamathematics.* D. Reidel, Dordrecht, Holland, 1980.

[**Min**] Minsky, M. L. *Computation : Finite and Infinite Machines.* Prentice-Hall, N.J., 1967.

[**Old1**] Olderog, E-R. *A Characterization of Hoare-like Calculi Based on Copy Rules.* Acta Inf. 16, 1981, 161-197.

[**Old2**] Olderog, E-R. *Sound and Complete Hoare's Logic For Programs with Pascal-like Procedures.* Proc. 15th ACM Symp, Theory of Computing, 1983, 320-329.

[**Pe**] Penrose, R. *The Emperor's New Mind.* Oxford U. Press, Oxford, 1989.

[**R**] Rogers, Hartley *Theory of Recursive Functions and Effective Computability.* McGraw-Hill, New York, 1967.

[**Ru-ed**] Rustin, Randall *Formal Semantics of Programming Languages.* Prentice-Hall, N.J., 1972.

[**S**] Schoenfield, J. *Recursion Theory.* Lecture Notes in Logic #1, Springer-Verlag, Berlin, Heidelberg, 1993.

[**Si**] Sipser, Michael *Introduction to the Theory of Computability.* PWS Publishing, Boston, 1997.

[**SHJM-ed**] Suppes, P., Henkin, L., Athanase, J., FloydMoisil, GR. C. *Logic, Methodology and Philosophy of Science IV.* North-Holland, Amsterdam, 1973.

[**Sö**] Stenlund, Sören *Combinators, λ-terms, and Proof Theory.* Reidel Pub. Co., Dordrecht, 1972.

[**St-ed**] Steel, T.B. *Formal Language Description Languages for Computer Programming.* North-Holland, Amsterdam, 1966.

[**St**] Stoy, Joseph *Denotational Semantics : the Scott-Strachey approach to programming language theory.* MIT Press, Cambridge, Mass., 1977.

[**SW**] Sommerhalder, R. and van Westrhenen, S. C. *The Theory of Computability.* Addison-Wesley, Wokingham, U.K., 1988.

[**W**] Willson, Ben *A* **BTEN***-command for Ackermann's Function.* preprint, University of Waterloo, 2002.

[**Wa**] Wadsworth, Christopher P. *The Relation between Computational and Denotational Properties for Scott's D_∞-Models of the Lambda-calculus.* SIAM J. Comput. 5(3) 1976, 488-521.