SIAM J. COMPUT.
Vol. 5, No. 3, September 1976

DATA TYPES AS LATTICES

To the Memory of Christopher Strachey, 1916–1975

DANA SCOTT†

Abstract. The meaning of many kinds of expressions in programming languages can be taken as elements of certain spaces of "partial" objects. In this report these spaces are modeled in one universal domain $\mathbf{P}\omega$, the set of all subsets of the integers. This domain renders the connection of this semantic theory with the ordinary theory of number theoretic (especially general recursive) functions clear and straightforward.

Key words. programming language semantics, lattice, continuous lattice, algebraic lattice, computability, retract, combinatory logic, lambda calculus, recursion theorem, enumeration degrees, continuous function, fixed-point theorem

Introduction. Investigations begun in 1969 with Christopher Strachey led to the idea that the denotations of many kinds of expressions in programming languages could be taken as elements of certain kinds of spaces of "partial" objects. As these spaces could be treated as function spaces, their structure at first seemed excessively complicated—even impossible. But then the author discovered that there were many more spaces than we had first imagined—even wanted. They could be presented as lattices (or as some prefer, semilattices), and the main technique was to employ topological ideas, in particular the notion of a continuous function. This approach and its applications have been presented in a number of publications, but that part of the foundation concerned with *computability* (in the sense of recursion theory) was never before adequately exposed. The purpose of the present paper is to provide such a foundation and to simplify the whole presentation by a de-emphasis of abstract ideas. An Appendix and the references provide a partial guide to the literature and an indication of connections with other work.

The main innovation in this report is to model everything within one "universal" domain $\mathbf{P}\omega = \{x | x \subseteq \omega\}$, the domain of all subsets of the set ω of nonnegative integers. The advantages are many: by the most elementary considerations $\mathbf{P}\omega$ is recognized to be a lattice and a topological space. In fact, $\mathbf{P}\omega$ is a *continuous lattice*, even an *algebraic lattice*, but in the beginning we do not even need to define such an "advanced" concept; we can save these ideas for an *analysis* of what has been done here in a more direct way. Next by taking the set of integers as the basis of the construction, the connection with the ordinary theory of number-theoretic (especially, general recursive) functions can be made clear and straightforward.

The model $\mathbf{P}\omega$ can be intuitively viewed as the domain of *multiple-valued* integers; what is new in the presentation is that functions are not only multiple-valued but also "multiple-argumented". This remark is given a precise sense in § 2 below, but the upshot of the idea is that multiple-valued integers are regarded as

* Received by the editors May 23, 1975, and in revised form March 17, 1976.
† Mathematics Department, Oxford University, Oxford, England OX2 6PE.

objects in themselves—possibly infinite—and as something more than just the collection of single integers contained in them. This combination of the finite with the infinite into a single domain, together with the idea that a continuous function can be reduced to its *graph* (in the end, a set of integers), makes it possible to view an $x \in \mathbf{P}\omega$ at one time as a value, at another as an argument, then as an integer, then as a function, and still later as a functional (or combinator). The "paradox" of self-application (as in $x(x)$) is solved by allowing the *same* x to be used in two *different* ways. This is done in ordinary recursion theory via Gödel numbers (as in $\{e\}(e)$), but the advantage of the present theory is that not only is the function concept the *extensional* one, but it includes *arbitrary* continuous functions and not just the computable ones.

Section 1 introduces the elementary ideas on the topology of $\mathbf{P}\omega$ and the continuous functions including the fixed-point theorem. Section 2 has to do with computability and definability. The language LAMBDA is introduced as an extension of the pure λ-calculus by four arithmetical combinators; in fact, it is indicated in § 3 how the whole system could be based on *one* combinator. What is shown is that computability in $\mathbf{P}\omega$ according to the natural definition (which assumes that we already know what a recursively enumerable set of integers is) is equivalent to LAMBDA-definability. The main tool is, not surprisingly, the first recursion theorem formulated with the aid of the so-called *paradoxical combinator* $\mathbf{Y}$. The plan is hardly original, but the point is to work out what it all means in the model.

Along the way we have to show how to give every λ-term a denotation in $\mathbf{P}\omega$; the resulting principles of λ-calculus that are thereby verified are summarized in Table 1. Of these the first three, (α), (β), and (ξ), are indeed valid in the model; however, rule (η), which is a stronger version of extensionality, *fails* in the $\mathbf{P}\omega$ model. This should not be regarded as a disadvantage since the import of (η) is to suppose every object is a function. A quick construction of these special models is indicated at the end of § 5. Since $\mathbf{P}\omega$ is partially ordered by $\subseteq$, there are also laws involving this relation. Law (ξ^*) is an improvement of (ξ); while (μ) is a form of monotonicity for application.

TABLE 1

Some laws of λ-calculus

(α)	$\lambda x.\tau = \lambda y.\tau[y/x]$
(β)	$(\lambda x.\tau)(y) = \tau[y/x]$
(ξ)	$\lambda x.\tau = \lambda x.\sigma$ iff $\forall x.\tau = \sigma$
(η)	$y = \lambda x.y(x)$
(ξ^*)	$\lambda x.\tau \subseteq \lambda x.\sigma$ iff $\forall x.\tau \subseteq \sigma$
(μ)	$x \subseteq y$ and $u \subseteq v$ imply $u(x) \subseteq v(y)$

Section 3 has to do with enumeration and degrees. Gödel numbers for LAMBDA are defined in a very easy way which takes advantage of the notation of combinators. This leads to the second recursion theorem, and results on incompleteness and undecidability follow along standard lines. *Relative recursiveness* is also very easy to define in the system, and we make the tie-in with *enumeration degrees* which correspond to finitely generated combinatory subalgebras of $\mathbf{P}\omega$. Finally a theorem of Myhill and Shepherdson is interpreted as a most satisfactory completeness property for definability in the system.

Sections 4 and 5 show how a calculus of *retracts* leads to quite simple definitions of a host of useful domains (as lattices). Section 6 investigates the classification of other subsets (nonlattices) of $\mathbf{P}\omega$; while § 7 contrasts partial (multiple-valued) functions with total functions, and interprets various theories of functionality. Connections with category theory are mentioned.

What is demonstrated in this work is how the language LAMBDA, together with its interpretation in $\mathbf{P}\omega$, is an extremely convenient vehicle for *definitions* of computable functions on complex structures (all taken as subdomains of $\mathbf{P}\omega$). It is a "high-level" programming language for recursion theory. It is *applied* combinatory logic, which in usefulness goes far beyond anything envisioned in the standard literature. What has been shown is how many interesting predicates can be *expressed* as equations between continuous functions. What is needed next is a development of the *proof theory* of the system along the lines of the work of Milner, which incorporates the author's extension of McCarthy's rule of recursion induction to this high-level language. Then we will have a flexible and practical "mathematical" theory of computation.

1. Continuous functions. The domain $\mathbf{P}\omega$ of all subsets of the set ω of nonnegative integers is a complete lattice under the partial ordering $\subseteq$ of set inclusion, as is well known. We use the usual symbols $\cup, \cap, \bigcup, \bigcap$ for the finite and infinite lattice operations of union and intersection. $\mathbf{P}\omega$ is of course also a Boolean algebra; and for complements we write

$$\sim x = \{n | n \notin x\}$$

where it is understood that such variables as i, j, k, l, m, n range over integers *in* ω, while u, v, w, x, y, z range over *subsets of* ω.

The domain $\mathbf{P}\omega$ can also be made into a topological space—in many ways. A common method is to match each $x \subseteq \omega$ with the corresponding characteristic function in $\{0, 1\}^{\omega}$, and to take the induced product topology. In this way $\mathbf{P}\omega$ is a totally disconnected compact Hausdorff space homeomorphic to the Cantor "middle-third" set. This is *not* the topology we want; it is a *positive-and-negative* topology which makes the function $\sim x$ continuous. We want a weaker topology: the topology of positive "information", which has the advantage that all continuous functions possess fixed points. (The equation $x = \sim x$ is impossible.) The topology that we do want is exactly that appropriate to considering $\mathbf{P}\omega$ to be a continuous lattice. But all this terminology of abstract mathematics is quite unnecessary, since the required definitions can be given in very elementary terms.

To make the topology "visible", we introduce a standard enumeration $\{e_n | n \in \omega\}$ of all *finite* subsets of ω. Specifically we set

$$e_n = \{k_0, k_1, \cdots, k_{m-1}\},$$

provided that $k_0 < k_1 < \cdots < k_{m-1}$ and $n = \sum_{i<m} 2^{k_i}$. Thus n is the code number for e_n, and the elements of e_n are the exponents in the binary expansion of the integer n. This is a one-to-one enumeration of finite subsets, where $k \in e_n$ always implies $k < n$, the function $\max(e_n)$ is (primitive) recursive in n, and the relations $k \in e_n$, $e_n \subseteq e_m$, $e_n = e_m \cup e_k$ are all (primitive) recursive in k, m, n.

Topologically speaking the finite sets e_n are *dense* in the space $\mathbf{P}\omega$, for each $x \in \mathbf{P}\omega$ is the "limit" of its finite subsets in the sense that

$$x = \bigcup \{e_n | e_n \subseteq x\}.$$

To make this precise we need a rigorous definition of *open subset* of $\mathbf{P}\omega$.

DEFINITION. A *basis* for the neighborhoods of $\mathbf{P}\omega$ consists of those sets of the form:

$$\{x \in \mathbf{P}\omega | e_n \subseteq x\},$$

for a given e_n. An arbitrary *open subset* is then a union of basic neighborhoods.

It is easy to prove that an open subset $U \subseteq \mathbf{P}\omega$ is just a set of "finite character"; that is, a set such that for all $x \in \mathbf{P}\omega$ we have $x \in U$ *if and only if* some finite subset of x also belongs to U. An alternate approach would define directly what we mean by a continuous function using the idea that such functions must preserve limits.

DEFINITION. A function $f: \mathbf{P}\omega \to \mathbf{P}\omega$ is *continuous* iff for all $x \in \mathbf{P}\omega$ we have:

$$f(x) = \bigcup \{f(e_n) | e_n \subseteq x\}.$$

Again it is an easy exercise to prove that a function is continuous in the sense of this definition iff it is continuous in the usual topological sense (namely: inverse images of open sets are open). For giving proofs it is even more convenient to have the usual ε-δ formulation of continuity.

THEOREM 1.1 (The characterization theorem). *A function $f: \mathbf{P}\omega \to \mathbf{P}\omega$ is continuous iff for all $x \in \mathbf{P}\omega$ and all ε_m we have:*

$$e_m \subseteq f(x) \quad \textit{iff} \quad \exists e_n \subseteq x . e_m \subseteq f(e_n).$$

Note that open sets and continuous functions have *a monotonicity property*:

whenever $x \subseteq y$ and $x \in U$, then $y \in U$; and
whenever $x \subseteq y$, then $f(x) \subseteq f(y)$.

This gives a precise expression to the "positive" character of our topology. However, note too that openness and continuity mean rather more than just monotonicity. In particular, a continuous function is completely determined by the pairs of integers such that $m \in f(e_n)$, as can be seen from the definition. (Hence, there are only a continuum number of continuous functions, but more than a continuum number of monotonic functions.) This brings us to the definition of the *graph* of a continuous function.

To formulate the definition, we introduce a standard enumeration (n, m) of all pairs of integers. Specifically we set

$$(n, m) = \tfrac{1}{2}(n + m)(n \dot{+} m + 1) + m.$$

This is the enumeration along the "little diagonals" going from left to right, and it produces the ordering:

$$(0, 0), (1, 0), (0, 1), (2, 0), (1, 1), (0, 2), (3, 0), (2, 1), \cdots.$$

Note that $n \leqq (n, m)$ and $m \leqq (n, m)$ with equality possible only in the cases of $(0, 0)$ and $(1, 0)$. This is a one-to-one enumeration, and the inverse functions are (primitive) recursive—but we do not require at the present any notation for them.

DEFINITION. The *graph* of a continuous function $f : \mathbf{P}\omega \to \mathbf{P}\omega$ is defined by the equation:

$$\mathbf{graph}(f) = \{(n, m) | m \in f(e_n)\};$$

while the *function* determined by any set $u \subseteq \omega$ is defined by the equation:

$$\mathbf{fun}(u)(x) = \{m | \exists e_n \subseteq x.(n, m) \in u\}.$$

THEOREM 1.2 (The graph theorem). *Every continuous function is uniquely determined by its graph in the sense that*:

(i) $$\mathbf{fun}\,(\mathbf{graph}(f)) = f.$$

Conversely, every set of integers determines a continuous function and we have:

(ii) $$u \subseteq \mathbf{graph}(\mathbf{fun}(u)),$$

where equality holds just in case u satisfies:

(iii) *whenever* $(k, m) \in u$ *and* $e_k \subseteq e_n$, *then* $(n, m) \in u$.

Besides functions of one variable we need to consider also functions of several variables. The official definition for one variable given above can be extended simply by saying $f(x, y, \cdots)$ is continuous iff it is continuous in *each* of $x, y, \cdots$. Those familiar with the product topology can prove that for our special positive topology on $\mathbf{P}\omega$ this is equivalent to being continuous on the product space (continuous in the variables *jointly*). Those interested only in elementary proofs can calculate out directly from the definition (with the aid of 1.1) that continuity behaves under combinations by substitution [as in: $f(g(x, y), h(y, x, y))$].

THEOREM 1.3 (The substitution theorem). *Continuous functions of several variables on* $\mathbf{P}\omega$ *are closed under substitution.*

The other general fact about continuous functions that we shall use constantly concerns fixed points, whose existence can be proved using a well-known method.

THEOREM 1.4 (The fixed-point theorem). *Every continuous function* $f : \mathbf{P}\omega \to \mathbf{P}\omega$ *has a least fixed point given by the formula*:

$$\mathbf{fix}(f) = \bigcup\{f^n(\varnothing) | n \in \omega\},$$

where $\varnothing$ *is the empty set and* f^n *is the n-fold composition of f with itself.*

Actually **fix** is a *functional* with continuity properties of its own. We shall not give the required definitions here because they can be more easily derived from the construction of the model given in the next section.

For those familiar with the abstract theory of topological spaces we give in conclusion two general facts about continuous functions with values in $\mathbf{P}\omega$ which indicate the scope and generality of our method.

THEOREM 1.5 (The extension theorem). *Let X and Y be arbitrary topological spaces where $X \subseteq Y$ as a subspace. Then every continuous function $f: X \to \mathbf{P}\omega$ can be extended to a continuous function $\bar{f}: Y \to \mathbf{P}\omega$ defined by the equation:*

$$\bar{f}(y) = \bigcup\{\bigcap\{f(x) | x \in X \cap U\} | y \in U\},$$

where $y \in Y$, and U ranges over the open subsets of Y.

THEOREM 1.6 (The embedding theorem). *Every T_0-space X with a countable basis $\{U_n | n \in \omega\}$ for its topology can be embedded in $\mathbf{P}\omega$ by the continuous function $\varepsilon: X \to \mathbf{P}\omega$ defined by the equation:*

$$\varepsilon(x) = \{n | x \in U_n\}.$$

Technically the T_0-hypothesis is what is needed to show that ε is one-to-one. The upshot of these two theorems is that in looking for (reasonable) topological structures we can confine attention to the subspaces of $\mathbf{P}\omega$ and to continuous functions defined on *all* of $\mathbf{P}\omega$. Thus the emphasis on a single space is justified structurally. What we shall see in the remainder of this work is that the use of a single space is also justified practically because the required subspaces and functions can be *defined* in very simple ways by a natural method of equations.

In order to make the plan of the work clearer, the proofs of the theorems have been placed in an Appendix when they are more than simple exercises.

2. Computability and definability. The purpose of the first section was to introduce in a simple-minded way the basic notions about the topology of $\mathbf{P}\omega$ and its continuous functions. In this section we wish to present the details of a powerful language for defining *particular* functions—especially computable functions—and initiate the study of the *use* of these functions. This study is then extended in different ways in the following sections.

Before looking at the language, a short discussion of the "meaning" of the elements $x \in \mathbf{P}\omega$ will be helpful from the point of view of motivation. Now in itself $x \in \mathbf{P}\omega$ is a *set*, but this does not reveal its *meaning*. Actually x has no "fixed" meaning, because it can be *used* in strikingly different ways; we look for meaning here solely in terms of use. Nevertheless it is possible to give some coherent guidelines.

In the first place it is convenient to let the singleton subsets $\{n\} \in \mathbf{P}\omega$ stand for the corresponding integers. In fact, we shall enforce by convention the *equation* $n = \{n\}$ as a way of simplifying notation. In this way, $\omega \subseteq \mathbf{P}\omega$ as a subset. (Note that our convention conflicts with such set-theoretical equations as $5 = \{0, 1, 2, 3, 4\}$. What we have done is to abandon the usual set-theoretical conventions in favor of a slight redefinition of *set of integers* which produces a more helpful convention for present purposes.) So, if we choose, a singleton "means" a single integer. The next

question is what a "large" set $x \in \mathbf{P}\omega$ could mean. Here is an answer: if we write:

$$x = \{0, 5, 17\} = 0 \cup 5 \cup 17,$$

we are thinking of x as a *multiple* integer. This is especially useful in the case of multiple-valued function where we can write:

$$f(a) = 0 \cup 5 \cup 17.$$

Then "$m \in f(a)$" can be interpreted as "m is *one* value of f at a." Now $a \in \mathbf{P}\omega$, too, and so it is a multiple integer also. This brings us to an important point.

A multiple integer is (often) *more* than just the (random) collection of its elements. From the definition of continuity, $m \in f(a)$ is equivalent to $m \in f(e_n)$ with $e_n \subseteq a$. We may not be able to reduce this to $m \in f(\{n\})$ with $n \in a$ without additional assumptions on f. Indeed we shall take advantage of the feature of continuous functions whereby the elements of an argument a can join in *cooperation* in determining $f(a)$. Needless to say, continuity implies that the cooperation cannot extend beyond *finite* configurations, and so we can say that a *is* the union (or limit) of its finite subsets. However, finitary cooperation will be found to be quite a powerful notion.

Where does this interpretation leave the empty set $\varnothing$? When we write "$f(a) = \varnothing$" we can read this as "f has *no* value at a", or "f is *undefined* at a". In this case $f(a)$ exists (as a set), but it is not "defined" as an *integer.* Single- (or singleton) valued functions are "well-defined", but multiple-valued functions are rather "over-defined".

How does this interpretation fit in with monotonicity? In case $a \subseteq b$ and $m \in f(a)$, then we must have $m \in f(b)$. We can read "$a \subseteq b$" as "b is an *improvement* of a" is *better-defined* than a". The point of monotonicity is that the better we define an argument, the better we define a value. 'Better" does not imply "well" (that is, singleton-valuedness), and overdefinedness may well creep in. This is not the fault of the function; it is our fault for not choosing a different function.

As a subspace $\omega \subseteq \mathbf{P}\omega$ is *discrete.* This implies that *arbitrary* functions $p : \omega \to \omega$ are *continuous.* Note that $p : \omega \to \mathbf{P}\omega$ as well, because $\omega \subseteq \mathbf{P}\omega$. By Theorem 1.5 we can extend p continuously to $\bar{p} : \mathbf{P}\omega \to \mathbf{P}\omega$. The formula given produces this function:

$$\bar{p}(x) = \begin{cases} \bigcap \{p(n) \mid n \in \omega\} & \text{if } x = \varnothing; \\ p(n) & \text{if } x = n \in \omega; \\ \omega & \text{otherwise.} \end{cases} \tag{2.1}$$

This is a rather abrupt extension of p (the maximal extension); a more gradual, continuous extension (the minimal extension) is determined by this equation:

$$\hat{p}(x) = \bigcup \{p(n) \mid n \in x\}. \tag{2.2}$$

The same formulae work for *all* multiple-valued functions $p : \omega \to \mathbf{P}\omega$. Functions like $f = \hat{p}$ are exactly characterized as being those continuous functions $f : \mathbf{P}\omega \to \mathbf{P}\omega$ which in addition are *distributive* in the sense of these equations:

$$f(x \cup y) = f(x) \cup f(y) \quad \text{and} \quad f(\varnothing) = \varnothing.$$

The sets $\varnothing$ and ω play special roles. When we consider them as *elements* of $\mathbf{P}\omega$ we shall employ the notation:

$$\bot = \varnothing \quad \text{and} \quad \top = \omega.$$

The element $\bot$ is the most "undefined" integer, and $\top$ is the most "overdefined". All others are in between.

One last general point on meaning: suppose $x \in \mathbf{P}\omega$ and $k \in x$. Then $k = (n, m)$ for suitable (uniquely determined) integers n and m. That is to say, every element of x can be regarded as an *ordered pair*; thus, x can be used as a *relation*. Such an operation as

$$x; y = \{(n, l) | \exists m.(n, m) \in x, (m, l) \in y\} \tag{2.3}$$

is then a continuous function of two variables that treats both x and y as relations. On the other hand we could define a quite different continuous function such as

$$x + y = \{n + m | n \in x, m \in y\} \tag{2.4}$$

which treats both x and y *arithmetically*. The only reason we shall probably never write $(x + y)$; x again is that the values of this perfectly well-defined continuous function are, for the most part, quite uninteresting. There is, however, no theoretical reason why we cannot use the same set with several different "meanings" in the same formula. Of course if we do so, it is to be expected that we will show the point of doing this in the special case. We turn now to the definition of the general language for defining all such functions.

The *syntax* and *semantics* of the language LAMBDA are set out in Table 2. The syntax is indicated on the left, and the meanings of the combinations are shown on the right as subsets of $\mathbf{P}\omega$. This is the basic language and could have been given (less understandably) in terms of combinators (see Theorem 2.4). It is, however, a very primitive language, and we shall require many definitions before we can see why such functions as in (2.3) and (2.4) are themselves definable.

TABLE 2
The language LAMBDA

$0 = \{0\}$
$x+1 = \{n+1 \mid n \in x\}$
$x-1 = \{n \mid n+1 \in x\}$
$z \supset x,\ y = \{n \in x \mid 0 \in z\} \cup \{m \in y \mid \exists k.k+1 \in z\}$
$u(x) = \{m \mid \exists e_n \subseteq x.(n, m) \in u\}$
$\lambda x.\tau = \{(n, m) \mid m \in \tau[e_n/x]\}$

The definition has been left somewhat informal in hopes that it will be more understandable. In the above, τ is any term of the language. LAMBDA is *type-free* and allows any combination to be made by substitution into the given functions. There is one primitive constant (0); there are two unary functions $(x+1, x-1)$; there is one binary function $(u(x))$ and one ternary function $(z \supset x, y)$; finally there is one variable binding operator $(\lambda x.\tau)$. The first three equations have obvious sense. In the fourth, $z \supset x, y$ is McCarthy's *conditional expression* (a test for *zero*). Next $u(x)$ defines *application* (u is treated as a graph and x as a set), and $\lambda x.\tau$ is *functional abstraction* (compare the definition of **fun**). In defining $\lambda x.\tau$, we use $\tau[e_n/x]$ as a shorthand for *evaluating* the term τ when the variable x is given the value e_n.

Note that the functions are all multiple-valued. Thus we have such a result as:

$$(6 \cup 10)+1 = 7 \cup 11. \tag{2.5}$$

The partial character of subtraction has expression as:

$$0-1 = \perp. \tag{2.6}$$

We shall see how to define + and − in LAMBDA later. The conditional could also have been defined by cases:

$$z \supset x, y = \begin{cases} \perp & \text{if } z = \perp; \\ x & \text{if } z = 0; \\ y & \text{if } 0 \notin z \neq \perp; \\ x \cup y & \text{if } 0 \in z \neq 0. \end{cases} \tag{2.7}$$

We say that a LAMBDA-term τ defines a function of its free variables (at least). Other results depend on this fundamental proposition:

THEOREM 2.1 (The continuity theorem). *All LAMBDA definable functions are continuous.*

Once that is proved, we can use Theorem 1.2 to establish:

THEOREM 2.2 (The conversion theorem). *The three basic principles* (α), (β), (ξ) *of* λ*-conversion are all valid in the model.*

By "model" here we of course understand the interpretation of the language where the semantics gives terms denotations in $\mathbf{P}\omega$ according to the stated definition. Through this interpretation, more properly speaking, $\mathbf{P}\omega$ becomes a model for the axioms (α), (β), (ξ). Two well-known results of the calculus of λ-conversion allow the reduction of functions of several variables to those of *one*, and the reduction of all the primitives to *combinators* (constants)—all this with the aid of the binary operation of application.

THEOREM 2.3 (The reduction theorem). *Any continuous function of* k*-variables can be written as*

$$f(x_0, x_1, \cdots, x_{k-1}) = u(x_1) \cdots (x_{k-1}),$$

where u *is a suitably chosen element of* $\mathbf{P}\omega$.

THEOREM 2.4 (The combinator theorem). *The LAMBDA-definable func-*

tions can be generated (from variables) by iterated application with the aid of these six constants:

$$\mathbf{0} = 0$$

$$\mathbf{suc} = \lambda x.x + 1$$

$$\mathbf{pred} = \lambda x.x - 1$$

$$\mathbf{cond} = \lambda x \lambda y \lambda z.z \supset x, y$$

$$\mathbf{K} = \lambda x \lambda y.x$$

$$\mathbf{S} = \lambda u \lambda v \lambda x.u(x)(v(x))$$

But the result that makes all this model building and combinatory logic *mathematically* interesting concerns the so-called *paradoxical combinator* defined by the equation:

$$\mathbf{Y} = \lambda u.(\lambda x.u(x(x)))(\lambda x.u(x(x))). \tag{2.8}$$

THEOREM 2.5 (The first recursion theorem). *If u is the graph of a continuous function f, then $\mathbf{Y}(u) = \mathbf{fix}(f)$, the least fixed point of f.*

There are two points to note here: the fixed point is LAMBDA-definable if f is; and $\mathbf{Y}$ defines a *continuous* operator. The word "recursion" is attached to the theorem because fixed points are employed to solve recursion equations. It would not be correct to call the fixed-point theorem (Theorem 1.4) the recursion theorem since it only shows that fixed points *exist* and not how they are *definable* in a language. The *second* recursion theorem (in Kleene's terminology) is related, but it involves Gödel numbers as introduced in § 3.

From this point on we see no need to distinguish continuous functions from elements of $\mathbf{P}\omega$; a continuous function will be *identified* with its graph. Note that u is a graph iff $u = \lambda x.u(x)$, which is equivalent to Theorem 1.2 (iii). For this reason (functions are graphs) we propose the name *Graph Model* for this model of the λ-calculus. (There is more to LAMBDA than just λ, however.)

The identification of functions with graphs entails that the function *space* of all continuous functions from $\mathbf{P}\omega$ into $\mathbf{P}\omega$ is to be identified (one-to-one) with the subspace

$$\mathbf{FUN} = \{u | u = \lambda x.u(x)\} \subseteq \mathbf{P}\omega.$$

The identification is *topological* in that the subspace topology agree with the product topology on the function space. This is the topology of pointwise convergence and is closely connected with the lattice structure on the function space which is also defined pointwise (that is, argumentwise). In the notation of λ-abstraction we can express this as the extension of the axiom of extensionality called (ξ^*) in Table 1 of the Introduction. The laws in Table 1 are not the only ones valid in the model, however. We may also note such argumentwise distribution

laws as:

$$(f \cup g)(x) = f(x) \cup g(x); \tag{2.9}$$

$$(\lambda x.\tau) \cup (\lambda x.\sigma) = \lambda x.(\tau \cup \sigma); \tag{2.10}$$

$$(f \cap g)(x) = f(x) \cap g(x); \tag{2.11}$$

$$(\lambda x.\tau) \cap (\lambda x.\sigma) = \lambda x.(\tau \cap \sigma). \tag{2.12}$$

In the above f and g must be graphs. It is also true that if $\mathscr{F} \subseteq \mathbf{P}\omega$, then

$$\bigcup\{f | f \in \mathscr{F}\}(x) = \bigcup\{f(x) | f \in \mathscr{F}\}, \tag{2.13}$$

but the same does not hold for $\bigcap$.

We state now a sequence of minor results which show why some simple functions and constants are LAMBDA-definable.

$$\bot = (\lambda x.x(x))(\lambda x.x(x)); \tag{2.14}$$

$$x \cup y = (\lambda z.0) \supset x, y; \qquad (\textit{Hint}: 0, 1 \in \lambda z.0) \tag{2.15}$$

$$\top = \mathbf{Y}(\lambda x.0 \cup (x+1)); \tag{2.16}$$

$$x \cap y = \mathbf{Y}(\lambda f \lambda x \lambda y.x \supset (y \supset 0, \bot), f(x-1)(y-1)+1)(x)(y). \tag{2.17}$$

The elements $\bot$ and $\top$ are graphs, by the way, and we can characterize them as the only fixed points of the combinator $\mathbf{K}$:

$$a = \lambda x.a \quad \text{iff} \quad a = \bot \quad \text{or} \quad a = \top. \tag{2.18}$$

Next we use the notation $\langle x_0, x_1, \cdots, x_{n-1}\rangle$ for the function $\hat{p}$ where $p : \omega \to \mathbf{P}\omega$ is defined by:

$$p(i) = \begin{cases} x_i & \text{if } i < n; \\ \bot & \text{if } i \geqq n. \end{cases}$$

$$\langle\rangle = \bot \tag{2.19}$$

$$\langle x\rangle = \lambda z.z \supset x, \bot \tag{2.20}$$

$$\langle x, y\rangle = \lambda z.z \supset x, (z-1 \supset y, \bot) \tag{2.21}$$

$$\langle x_0, x_1, \cdots, x_n\rangle = \lambda z.z \supset x_0, \langle x_1, \cdots, x_n\rangle(z-1). \tag{2.22}$$

Obviously we should formalize the subscript notation so that $u_x = u(x)$; then we find:

$$\langle x_0, x_1, \cdots, x_{n-1}\rangle_i = \begin{cases} x_i & \text{if } i < n; \\ \bot & \text{if } i \geq n. \end{cases} \tag{2.23}$$

This gives us the method of LAMBDA-defining *finite* sequences (in a quite natural way), and the next step is to consider *infinite* sequences. But these are just the functions $\hat{p}$ where $p : \omega \to \mathbf{P}\omega$ is arbitrary. What we need then is a condition expressible in the language equivalent to saying $u = \hat{p}$ for some p. This is the same as

$$u = \lambda x.\bigcup\{u_i | i \in x\},$$

but the $\bigcup$- and set-notation is not part of LAMBDA. We are forced into a recursive definition:

$$\$ = \mathbf{Y}(\lambda s \lambda u \lambda z.z \supset u_0, s(\lambda t.u_{t+1})(z-1)). \tag{2.24}$$

This equation generalizes (2.22) and we have:

$$\$(u) = \lambda x.\bigcup\{u_i | i \in x\}. \tag{2.25}$$

Thus the combinator \$ "revalues" an element as a distributive function. This suggests introducing the λ-notation for such functions by the equation:

$$\lambda n \in \omega.\tau = \$(\lambda z.\tau[z/n]). \tag{2.26}$$

With all these conventions LAMBDA-notation becomes very much like ordinary mathematical notation without too much strain.

Suppose that f is any continuous function and $a \in \mathbf{P}\omega$. We can define $p : \omega \to \mathbf{P}\omega$ in the ordinary way by *primitive recursion* where:

$$p(0) = a;$$
$$p(n+1) = f(n)(p(n)).$$

The question is: can we give a LAMBDA-definition for $\hat{p}$ (in terms of f and a as constants, say)? The answer is clear, for we can prove:

$$\hat{p} = \mathbf{Y}(\lambda u \lambda n \in \omega.n \supset a, f(n-1)(u(n-1))). \tag{2.27}$$

This already shows that a large part of the definitions of recursion theory can be given in this special language. Of course, *simultaneous* (primitive) recursions can be transcribed into LAMBDA with the aid of the ordered tuples of (2.22), (2.23) above. But we can go further and connect with partial (and general) recursive functions. We state first a definition.

DEFINITION. A continuous function f of k-variables is *computable* iff the relationship

$$m \in f(e_{n_0})(e_{n_1}) \cdots (e_{n_{k-1}})$$

is recursively enumerable (r.e.) in the integer variables $m, n_0, n_1, \cdots, n_{k-1}$.

If q is a *partial* recursive function in the usual sense, then we can regard it as a mapping $q : \omega \to \omega \cup \{\perp\}$, where $q(n) = \perp$ means that q is undefined at n. Saying that q is partial *recursive* is just to say that $m \in q(n)$ is r.e. as a relation in n and m. It is easy to see that this is in turn equivalent to the recursive enumerability of the relationship $m \in \hat{q}(e_n)$; and so our definition is formally a generalization of the usual one. But it is also intuitively reasonable. To "compute" $y = f(x)$, in the one-variable case, we proceed by enumeration. First we begin the enumeration of all finite subsets $e_n \subseteq x$. For each of these f starts up an enumeration of the set $f(e_n)$; so we sit back and observe which $m \in f(e_n)$ by enumeration. The totality of all such m for all $e_n \subseteq x$ forms in the end the set y.

THEOREM 2.6 (The definability theorem). *For a k-ary continuous function f, the following are equivalent*:

(i) *f is computable*;

(ii) $\lambda x_0 \lambda x_1 \cdots \lambda x_{k-1}.f(x_0)(x_1) \cdots (x_{k-1})$ *as a set is r.e.*;

(iii) $\lambda x_0 \lambda x_1 \cdots \lambda x_{k-1}.f(x_0)(x_1) \cdots (x_{k-1})$ *is LAMBDA-definable.*

As a hint for the proof we may note that the method of (2.27) shows that all primitive recursive functions p have the corresponding $\hat{p}$ LAMBDA-definable. Next we remark that a nonempty r.e. set is the range of a primitive recursive function; but the range of p is $\hat{p}(\top)$, which is clearly LAMBDA-definable. That any LAMBDA-definable set (graph) is r.e. is obvious from the definition of the language itself. More details are given in the Appendix.

We may draw some interesting conclusions from the definability theorem. In the first place, we see that the countable collection $\mathbf{RE} \subseteq \mathbf{P}\omega$ of r.e. sets is *closed* under application and LAMBDA-definability. Indeed it forms a model for the λ-calculus (axioms (α), (β), (ξ^*) at least) and it also contains the arithmetical combinators. (Clearly there will be *many* intermediate submodels.) In the second place, we can see now how very easy it is to interpret λ-calculus in ordinary arithmetical recursion theory by means of quite elementary operations on r.e. sets. Thus the equivalence of λ-definability with partial recursiveness seems not to be all that good a piece of evidence for Church's Thesis. In his 1936 paper (a footnote on p. 346) Church says about λ-definability:

> The fact, however, that two such widely different and (in the opinion of the author) equally natural definitions of effective calculability turn out to be equivalent adds to the strength of the reasons adduced below for believing that they constitute as general a characterization of this notion as is consistent with the usual intuitive understanding of it.

The point never struck the present author as an especially telling one, and the reduction of λ-calculus to r.e. theory shows that the divergence between the theories is not at all wide. Of course it is a pleasant surprise to see how many complicated things can be defined in *pure* λ-calculus (without arithmetical combinators), but this fact cuts the wrong way as evidence for the thesis (we want stronger theories, not weaker ones). Post systems (or even first-order theories) are much better to mention in this connection, since they are obviously *more* inclusive in giving enumerations than Turing machines or Herbrand–Gödel recursion equations. But the equivalence proofs are all so easy! What one would like to see is a "natural" definition where the equivalence with r.e. is not just a mechanical exercise involving a few tricks of coding.

In the course of the development in this section we have stated many equations which are not found in Table 1, and which involve new combinators. In conclusion we would like to mention an equation about $\mathbf{Y}$ which holds in the model, which can be stated in pure λ-calculus, and which cannot be proved by ordinary reduction (though we shall not try to justify this last statement here). In order to shorten the calculations, we note from definition (2.8) that $\mathbf{Y}(u) = \mathbf{Y}(\lambda y.u(y))$; so by Theorem 2.5 this also equals $u(\mathbf{Y}(u))$.

$$\mathbf{Y}(\lambda f \lambda x.g(x)(f(x))) = \lambda x.\mathbf{Y}(g(x)). \tag{2.28}$$

Call the left side f' and the right f''. Now

$$f'' = \lambda x.g(x)(\mathbf{Y}(g(x)) = \lambda x.g(x)(f''(x)),$$

thus $f' \subseteq f''$, because f' is a least fixed point. On the other hand $f' = \lambda x.g(x)(f'(x))$, so $f'(x) = g(x)(f'(x))$. Thus $f''(x) \subseteq f'(x)$, because $f''(x)$ is a least fixed point. As this holds for all x, we see that $f'' \subseteq f'$; and so they are equal. There must be many other such equations.

3. Enumeration and degrees. A great advantage of the combinators from the formal point of view is that (bound) variables are eliminated in favor of "algebraic" combinations. The disadvantage is that the algebra is not all that pretty, as the combinations tend to get rather long and general laws are rather few. Nevertheless as a technical device it is mildly remarkable that we can have a notation for all r.e. sets requiring so few primitives. In the model defined here the reduction to one combinator rests on a lemma about conditionals:

$$\mathbf{cond}(x)(y)(\mathbf{cond}(x)(y)) = y. \tag{3.1}$$

Recall that **cond** (or $\supset$) is a test for zero, so that:

$$\mathbf{cond}(x)(y)(0) = x. \tag{3.2}$$

This suggests that we lump all combinators of Theorem 2.4 into this one:

$$\mathbf{G} = \mathbf{cond}(\langle \mathbf{suc}, \mathbf{pred}, \mathbf{cond}, \mathbf{K}, \mathbf{S}\rangle)(0). \tag{3.3}$$

We can then readily prove:

THEOREM 3.1 (The generator theorem). *All LAMBDA-definable elements can be obtained from* **G** *by iterated application.*

A distributive function f is said to be *total* iff $f(n) \in \omega$ for all $n \in \omega$. As they come from obvious primitive recursive functions, we do not stop to write out LAMBDA-definitions of these three total functions:

$$\mathbf{apply} = \lambda n \in \omega.\lambda m \in \omega.(n, m) + 1 \tag{3.4}$$

$$\mathbf{op}((n, m)) = n \tag{3.5}$$

$$\mathbf{arg}((n, m)) = m. \tag{3.6}$$

The point of these auxiliary combinators concerns our Gödel numbering of the r.e. sets. The number 0 will correspond to the generator **G**; while $(n, m)+1$ will correspond to the application of the nth set to the mth. This is formalized in the combinator **val** which is defined as the least fixed point of the equation:

$$\mathbf{val} = \lambda k \in \omega.k \supset \mathbf{G}, \mathbf{val}(\mathbf{op}(k-1))(\mathbf{val}(\mathbf{arg}(k-1))). \tag{3.7}$$

This function accomplishes the enumeration as follows:

THEOREM 3.2 (The enumeration theorem). *The combinator* **val** *enumerates the LAMBDA-definable elements in that* $\mathbf{RE} = \{\mathbf{val}(n) | n \in \omega\}$. *Further:*

(i) $\mathbf{val}(0) = \mathbf{G}$,

(ii) $\mathbf{val}(\mathbf{apply}(n)(m)) = \mathbf{val}(n)(\mathbf{val}(m))$.

As a principal application of the Enumeration Theorem we may mention the following: suppose u is *given* as LAMBDA-definable. We look at its definition and rewrite it in terms of combinators—eventually in terms of **G** alone. Then using 0 and apply we write down the name of an integer corresponding to the combination—say, n. By Theorem 3.2 we see that we have *effectively found* from

the definition an integer such that $\mathbf{val}\, n = u$. This remark can be strengthened by some numerology.

(3.8) $\mathbf{apply}(0)(0) = 1$ and $\mathbf{val}(1) = 0$;

(3.9) $\mathbf{apply}(0)(1) = 3$ and $\mathbf{val}(3) = \langle \mathbf{suc}, \cdots \rangle$;

(3.10) $\mathbf{apply}(3)(1) = 12$ and $\mathbf{val}(12) = \mathbf{suc}$.

Thus, define as the least fixed point:

(3.11) $\mathbf{num} = \lambda n \in \omega . n \supset 1, \mathbf{apply}(12)(\mathbf{num}(n-1))$,

and derive the equation for all $n \in \omega$:

(3.12) $\mathbf{val}(\mathbf{num}(n)) = n$.

We note that **num** is a primitive recursive (total) function. The combinator **num** allows us now to effectively find a LAMBDA-definition, corresponding to a given LAMBDA-definable element u, of an element v such that uniformly in the integer variable n we have $\mathbf{val}(v(n)) = u(n)$. Further, v is a primitive recursive (total) function. This is the technique involved in the proof of Kleene's well-known result:

THEOREM 3.3 (The second recursion theorem). *Take a LAMBDA-definable element v such that*:

(i) $\mathbf{val}(v(n)) = \lambda m \in \omega . \mathbf{val}(n)(\mathbf{apply}(m)(\mathbf{num}(m)))$,

and then define a combinator by:

(ii) $\mathbf{rec} = \lambda n \in \omega . \mathbf{apply}(v(n))(\mathbf{num}(v(n)))$.

Then we have a primitive recursive function with this fixed-point property:

(iii) $\mathbf{val}(\mathbf{rec}(n)) = \mathbf{val}(n)(\mathbf{rec}(n))$.

Note that if u is LAMBDA-definable, then we find first an n such that $\mathbf{val}(n) = u$. Next we calculate $k = \mathbf{rec}(n)$. This effectively gives us an integer such that $\mathbf{val}(k) = u(k)$. Gödel numbers represent expressions (combinations in **G**), and **val** maps the numbers to the values denoted by the expressions in the model. The k just found thus represents an expression whose value is defined in terms of its *own* Gödel number. In recursion theory there are many applications of this result. Another familiar argument shows:

THEOREM 3.4 (The incompleteness theorem). *The set of integers n such that* $\mathbf{val}(n) = \bot$ *is not r.e.; hence, there can be no effectively given formal system for enumerating all true equations between LAMBDA-terms.*

(A critic may sense here an application of Church's thesis in stating the metatheoretic consequence of the nonresult.) A few details of the proof can be given to see how the notation works. First let v be a (total) primitive recursive function such that:

$$\mathbf{val}(v(n)) = n \cap \mathbf{val}(n).$$

and note that:

$$n \cap \mathbf{val}(n) = \bot \quad \text{iff} \quad n \notin \mathbf{val}(n).$$

Call the set in question in Theorem 3.4 the set b. If it were r.e., then so would be:

$$\{n \in \omega | v(n) \in b\} = (\lambda n \in \omega . v(n) \cap b \supset n, n)(\top).$$

That would mean having an integer k such that:

$$\mathbf{val}(k) = \{n \in \omega | v(n) \in b\}.$$

But then:

$$\begin{aligned} k \in \mathbf{val}(k) \quad &\text{iff} \quad v(k) \in b \\ &\text{iff} \quad \mathbf{val}(v(k)) = \bot \\ &\text{iff} \quad k \notin \mathbf{val}(k), \end{aligned}$$

which gives us a contradiction. This is the usual diagonal argument.

The relationship $\mathbf{val}(n) = \mathbf{val}(m)$ means that the expressions with Gödel numbers n and m have the *same* value in the model. (This is not only not r.e. but is a complete Π^0_2-predicate.) A total mapping can be regarded as a *syntactical* transformation on expressions defined via Gödel numbers. Such a mapping p is called *extensional* if it has the property:

$$\mathbf{val}(p(n)) = \mathbf{val}(p(m)) \quad \textit{whenever} \quad \mathbf{val}(n) = \mathbf{val}(m).$$

The Myhill–Shepherdson theorem shows that extensional, syntactical mappings really depend on the *values* of the expressions. Precisely we have:

THEOREM 3.5 (The completeness theorem for definability). *If a (total) extensional mapping p is LAMBDA-definable, then there is a LAMBDA-definable q such that* $\mathbf{val}(p(n)) = q(\mathbf{val}(n))$ *for all* $n \in \omega$.

Of course q is uniquely determined (because the values of q are given at least on the finite sets). Thus any attempt to define something *new* by means of some strange mapping on Gödel numbers is bound to fail as long as it is *effective* and *extensional.* The main part of the argument is concentrated on showing these mappings to be *continuous*; that is why q exists.

The preceding results indicate that the expected results on r.e. sets are forthcoming in a smooth and unified manner in this setting. Some knowledge of r.e. theory was presupposed, but analysis shows that the knowledge required is slight. The notion of primitive recursive functions should certainly be well understood together with standard examples. Partial functions need not be introduced separately since they are naturally incorporated into LAMBDA (the theory of multiple-valued functions). As a working definition of r.e. one can take either "empty or the range of a primitive recursive function" or, more uniformly, "a set of the form $\{m | \exists n . m + 1 = p(n)\}$ where p is primitive recursive". A few obvious closure properties of r.e. sets should then be proved, and then an adequate foundation for the discussion of LAMBDA will have been provided. The point of introducing LAMBDA is that further closure properties are more easily expressed in a theory where equations can be variously interpreted as involving numbers, functionals, etc., without becoming too heavily involved in intricate Gödel numbering and encodings. Another useful feature of the present theory concerns the ease with which we can introduce *relative recursiveness.*

As we have seen, $\{\mathbf{val}(n)|n \in \omega\}$ is an enumeration of all r.e. sets. Suppose we add a new set a as a new *constant.* What are the sets enumerable in a? Answer: $\{\mathbf{val}(n)(a)|n \in \omega\}$, since in combinatory logic a parameter can always be factored out as an extra argument. Another way to put the point is this: for b to be *enumeration reducible* to a it is necessary and sufficient that $b = u(a)$ where $u \in \mathbf{RE}$. This is *word for word* the definition given by Rogers (1967, pp. 146–147). What we have done is to put the theory of enumeration operators (Friedberg–Rogers and Myhill–Shepherdson) into a general setting in which the language LAMBDA not only provides definitions but also the basis of a calculus for demonstrating properties of the operators defined. The algebraic style of this language throws a little light on the notion of enumeration degree. In the first place we can identify the degree of an element with the set of all objects reducible to it (rather than just those equivalent to it) and write

$$\mathbf{Deg}(a) = \{u(a)|u \in \mathbf{RE}\}.$$

The set-theoretical inclusion is then the same as the partial ordering of degrees. What kind of a partially ordered set do we have?

THEOREM 3.6 (The subalgebra theorem). *The enumeration degrees are exactly the finitely generated combinatory subalgebras of* $\mathbf{P}\omega$.

By "subalgebras" here we of course mean subsets containing $\mathbf{G}$ and closed under application (hence, they contain all of $\mathbf{RE}$, the least subalgebra). Part of the assertion is that every *finitely* generated subalgebra has a *single* generator (under application). This fact is an easy extension of Theorem 3.1. Not very much seems to be known about enumeration degrees. *Joins* can obviously be formed using the pairing function $\langle x, y\rangle$ on sets. Each degree is a countable set; hence, it is trivial to obtain the existence of a sequence of degrees whose *infinite join* is not a degree (not finitely generated). The intersection of subalgebras is a subalgebra—but it may not be a degree even starting with degrees. There are no minimal degrees above $\mathbf{RE}$, but there are minimal *pairs* of degrees. Also for a given degree there are only countably many degrees minimal over it; but the question of whether the partial ordering of enumeration degrees is dense seems still to be open.

Theorem 3.6 shows that the semilattice of enumeration degrees is naturally extendable to a *complete* lattice (the lattice of all subalgebras of $\mathbf{P}\omega$), but whether there is anything interesting to say about this complete lattice from the point of view of structure is not at all clear. Rogers has shown (1967, pp. 151–153) that Turing degrees can be defined in terms of enumeration degrees by restricting to special elements. In our style of notation we would define the space:

$$\mathbf{TOT} = \{u|u = \$(u) \text{ and } \forall n \in \omega.u(n) \in \omega\},$$

the space of all graphs of total functions. Then the system $\{\mathbf{Deg}(u)|u \in \mathbf{TOT}\}$ is isomorphic to the system of Turing degrees. Now there are many other interesting subsets of $\mathbf{P}\omega$. Whether the degree structure of these various subsets is worth investigation is a question whose answer awaits some new ideas.

Among the subsets of $\mathbf{P}\omega$ with natural mathematical structure, we of course have $\mathbf{FUN}$, which is a semigroup under $\circ = \lambda u \lambda v \lambda x.u(v(x))$. It is, however, a rather

complicated semigroup. We introduce for its study three new combinators:

(3.13) $\mathbf{R} = \lambda x.\langle 0, x\rangle;$

(3.14) $\mathbf{L} = \lambda x.x_1(x_2);$

(3.15) $\bar{u} = \lambda x.x_0 \supset \langle 1, u, x_1\rangle, \overline{u(x_1)}(x_2).$

THEOREM 3.7 (The semigroup theorem). *The countable semigroup* $\mathbf{RE} \cap \mathbf{FUN}$ *of computable enumeration operators is finitely generated by* $\mathbf{R}$, $\mathbf{L}$ *and* $\bar{\mathbf{G}}$.

The proof rests on the verification of two equations which permit an application of Theorem 3.1:

(3.16) $\mathbf{L} \circ \bar{u} \circ \mathbf{R} = \lambda x.u(x)$

(3.17) $\bar{u} \circ \bar{v} \circ \mathbf{R} = u(v).$

Certainly the word problem for $\mathbf{RE} \cap \mathbf{FUN}$ is unsolvable, indeed, not even recursively enumerable. Can the semigroup be generated by *two* generators by the way?

4. Retracts and data types. Data can be structured in many ways: ordered tuples, lists, arrays, trees, streams, and even operations and functions. The last point becomes clear if one thinks of *parameters*. We would normally hardly consider the pairing function $\lambda x\lambda y.\langle x, y\rangle$ as being in itself a piece of data. But if we treat the first variable as a parameter, then it can be specialized to a *fixed value*, say the element a, producing the function $\lambda y.\langle a, y\rangle$. This function is more likely to be the output of some process and *in itself* can be considered as a datum. It is rather like one *whole row* of a matrix. If we were to regard a two-argument function f as being a matrix, then its ath row would be exactly $\lambda y.f(a)(y)$. If s were a selection function, then, for example, $\lambda y.f(s(y))(y)$ would represent the selection of one element out of each column of the matrix. This selection could be taken as a specialization of parameters in the operator $\lambda u\lambda v\lambda y.u(v(y))(y)$. We have not been very definite here about the exact nature of the fixed a, f, or s, or the range of the variable y or the range of values of the function f. The point is only to recall a few elements of structure and to suggest an abstract view of data going beyond the usual iterated arrays and trees.

What then is a *data type*? Answer: a type of data. That is to say, a collection of data that have been grouped together for reasons of similarity of structure or perhaps mere convenience. Thus the collection may very well be a mixed bag, but more often than not canons of taste or demands of simplicity dictate an adherence to regularity. The grouping may be formed to eliminate irrelevant objects and focus the attention in other ways. It is frequently a matter of good organization that aids the understanding of complex definitions. In programming languages, one of the major reasons for making an explicit declaration of a data type (that is, the restriction of certain variables to certain "modes") is that the computed objects of that type can enjoy a special *representation* in the machine that allows the manipulation of these objects via the chosen representation to be reasonably efficient. This is a very critical matter for good language design and good compiler writing. In this report, however, we cannot discuss the problems of representation,

important as they may be. Our objective here is conceptual organization, and we wish to show how such ideas, in the language for computable functions used here, can find the proper expression.

Which are the data types that can be defined in LAMBDA? No final answer can be given since the number is infinite and inexhaustible. From one point of view, however, there is only one: $\mathbf{P}\omega$ itself. It is the universal type and all other types are subtypes of it; so $\mathbf{P}\omega$ plays a primary role in this exposition. But in a way it is too big, or at least too complex, since each of its elements can be used in so many different ways. When we specify a subtype the intention is to restrict attention to a special use. But even the various subtypes overlap, and so the same elements still get different uses. Style in writing definitions will usually make the differentiation clear though. The main innovation to be described in this section is the use of LAMBDA expressions to define types *as well as* elements. Certain expressions define *retracts* (or better: retraction mappings), and it is the *ranges* (or as we shall see: *sets of finite points*) of such retracts that form the groupings into types. Thus LAMBDA provides a calculus of type definitions including *recursive* type definitions. Examples will be explained both here and in the following sections. Note that types as retracts turn out to be types as *lattices*, that is, types of partial and many-valued objects. The problem of cutting these lattice types down to the perfect or complete objects is discussed in § 6. Another view of types and functionality of mappings is presented in § 7.

The notion of a retract comes from (analytic) topology, but it seems almost an accident that the idea can be applied in the present context. The word is employed not because there is some deep tie-up with topology but because it is short and rather descriptive. Three easy examples will motivate the general plan:

$$\textbf{fun} = \lambda u \lambda x.u(x); \tag{4.1}$$

$$\textbf{pair} = \lambda u.\langle u_0, u_1\rangle; \tag{4.2}$$

$$\textbf{bool} = \lambda u.u \trianglerighteq 0, 1. \tag{4.3}$$

Here $\trianglerighteq$ is the *doubly strict* conditional defined by

$$z \trianglerighteq x, y = z \supset (z \supset x, \top), (z \supset \top, y), \tag{4.4}$$

which has the property that if z is both zero and positive, then it takes the value $\top$ instead of the value $x \cup y$.

DEFINITION. An element $a \in \mathbf{P}\omega$ is called a *retract* iff it satisfies the equation $a = a \circ a$.

Of course the $\circ$-notation is used for functional composition in the standard way:

$$u \circ v = \lambda x.u(v(x)). \tag{4.5}$$

And it is quite simple to prove that each of the three combinators in (4.1)–(4.3) is a retract according to the definition. But what is the point?

Consider **fun**. No matter what $u \in \mathbf{P}\omega$ we take, $\textbf{fun}(u)$ is (the graph of) a *function*. And if u already is (the graph of) a function, then $u = \textbf{fun}(u)$. That is to say, the *range* of **fun** is the same as the set of *fixed points* of **fun** is the same as the set of all (graphs of) functions. Any mapping a whose range and fixed-point set

coincide satisfies $a = a \circ a$, and conversely. A retract is a mapping which "retracts" the whole space onto its range *and* which is the identity mapping on its range. That is the import of the equation $a = a \circ a$. Strictly speaking, the range is the retract and the mapping is the *retraction*, but for us the mapping is more important. (Note, however, that distinct retracts can have the same range.) We let the mapping stand in for the range.

Thus the combinator **fun** represents in itself the concept of a *function* (continuous function on $\mathbf{P}\omega$ into $\mathbf{P}\omega$). Similarly **pair**, represents the idea of a *pair* and **bool** the idea of being a boolean value as an element of $\{\bot, 0, 1, \top\}$, since we must think in the multiple-valued mode. What is curious (and, as we shall see, useful) is that all these retracts which are defining *subspaces* are at the same time *elements* of $\mathbf{P}\omega$.

DEFINITION. If a is a retract, we write $u : a$ for $u = a(u)$ and $\lambda u : a.\tau$ for $\lambda u.\tau[a(u)/u]$.

Since retracts are *sets* in $\mathbf{P}\omega$, we cannot use the ordinary membership symbol to signify that u *belongs* to the range of a; so we write $u : a$. The other notation with the λ-operator *restricts* a function to the range of a. For f to be so restricted simply means $f = f \circ a$. For the *range* of f to be *contained in* that of the retract a means $f = a \circ f$. These algebraic equations will be found to be quite handy. We are going to have a calculus of retracts *and* mappings between them involving many operators on retracts yet to be discovered. Before we turn to this calculus, we recall the well-known connection between lattices and fixed points.

THEOREM 4.1 (The lattice theorem). *The fixed points of any continuous function form a complete lattice (under* $\subseteq$*); while those of a retract form a continuous lattice.*

We note further that by the embedding theorem (Theorem 1.6), it follows that any separable (by which we mean countably-based) continuous lattice is a retract of $\mathbf{P}\omega$; hence, our universal space is indeed rich in retracts. A very odd point is that $a = a \circ a$ is a fixed-point equation itself ($\lambda u.u \circ u$ is obviously continuous). Thus the retraction mappings form a complete lattice. Is this a continuous lattice? (Ershov has proved it is not; see the Appendix for a sketch.) A related question is solved positively in the next section. Actually the ordering of retracts under $\subseteq$ does not seem to be all that interesting; a more algebraic ordering is given by:

DEFINITION. For retracts a and b we write $a \lessdot b$ for $a = a \circ b = b \circ a$.

The idea here should be clear: $a \lessdot b$ means that a is a retract *of* b. It is easy to prove the:

THEOREM 4.2 (The partial ordering theorem). *The retracts are partially ordered by* $\lessdot$.

There do not seem to be any lattice properties of $\lessdot$ of a general nature. Note, however, that if retracts *commute*, $a \circ b = b \circ a$, then $a \circ b$ is the greatest lower bound under $\lessdot$ of a and b. Also if we have a sequence where both $a_n \lessdot a_{n+1}$ and $a_n \subseteq a_{n+1}$ for all $n \in \omega$, then $\bigcup\{a_n | n \in \omega\}$ is the upper bound for the a_n under $\lessdot$, as can easily be argued from the definition by continuity of $\circ$.

Certainly there is no "least" retract under $\lessdot$. One has $\bot = \bot \circ a$ (recall: $\bot = \lambda x.\bot$), but not $a \circ \bot = \bot$. This last equation means more simply that $a(\bot) = \bot$; that is, a is *strict*. For retracts strictness is thus equivalent to $\bot \lessdot a$, so we can say that there is a least strict retract. The combinator $\mathbf{I} = \lambda u.u$ clearly represents the

largest retract (the *whole* space), and it is strict also. In a certain sense strictness can be assumed without loss of generality. For if a is not strict, let

$$b = \lambda x.\{n | a(x) \neq a(\bot)\}.$$

This function takes values in $\{\bot, \top\}$ and is continuous because $\{x \in \mathbf{P}\omega | a(x) \neq a(\bot)\}$ is *open*. Next define:

$$a^* = \lambda u.a(u) \cap b(u).$$

This is a strict retract whose range is homeomorphic (and lattice isomorphic) to that of a. Note, however, that the mapping from a to a^* is not continuous (or even monotonic).

To have a more uniform notation for retracts we shall often write **nil**, **id**, and **seq** for the combinators $\bot$, **I**, \$. Two further retracts of interest are

$$\textbf{open} = \lambda u.\{m | \exists e_n \subseteq e_m . n \in u\}; \tag{4.6}$$

$$\textbf{int} = \lambda u.u \sqsupseteq 0, \textbf{int}(u-1) \sqsupseteq u, u. \tag{4.7}$$

The range of **open** is lattice isomorphic to the lattice of open subsets of $\mathbf{P}\omega$; definition (4.6) is not a LAMBDA-definition of the retract, but such can be given.

In (4.7) we intend **int** to be the least fixed point of the equation. By induction on the least element of u (if any) one proves that:

$$\textbf{int}(u) = \begin{cases} \bot & \text{if } u = \bot; \\ u & \text{if } u \in \omega; \\ \top & \text{otherwise.} \end{cases}$$

This retract wipes out the distinctions between multiple values, moving all above the singletons up to $\top$; its range thus has a very simple structure. The retract **int** clearly generalizes **bool**. The range of **fun** is homeomorphic to the space of all continuous functions from $\mathbf{P}\omega$ into $\mathbf{P}\omega$; the range of **pair**, to the space of all ordered pairs; the range of **seq**, to the space of all infinite sequences. A combination like $\lambda u.\textbf{int} \circ \textbf{seq}(u)$ is a retract whose range is homeomorphic to the space of infinite sequences of elements from the range of **int**.

We now wish to introduce some operators that provide systematic ways of forming new combinations of retracts. There are three principal ones:

$$a \rightarrowtail b = \lambda u.b \circ u \circ a; \tag{4.8}$$

$$a \otimes b = \lambda u.\langle a(u_0), b(u_1)\rangle; \tag{4.9}$$

$$a \oplus b = \lambda u.u_0 \sqsupseteq \langle 0, a(u_1)\rangle, \langle 1, b(u_1)\rangle. \tag{4.10}$$

These equations clearly generalize (4.1)–(4.3). Before we explain our operators, note these three equations which hold for *arbitrary* $a, b, a', b' \in \mathbf{P}\omega$:

$$(a \rightarrowtail b) \circ (a' \rightarrowtail b') = (a' \circ a) \rightarrowtail (b \circ b'); \tag{4.11}$$

$$(a \otimes b) \circ (a' \otimes b') = (a \circ a') \otimes (b \circ b'); \tag{4.12}$$

$$(a \oplus b) \circ (a' \oplus b') = (a \circ a') \oplus (b \circ b'). \tag{4.13}$$

The reversal of order $(a' \circ a)$ on the right-hand side of (4.11) should be remarked.

These equations will be used not only for properties of types (ranges of retracts) but also for the mappings between the types.

THEOREM 4.3 (The function space theorem). *Suppose a, b, a', b', c are retracts. Then we have*:

(i) $a \multimap b$ *is a retract, and it is strict if* b *is*;
(ii) $v : a \multimap b$ *iff* $u = \lambda x : a.u(x)$ *and* $\forall x : a.u(x) : b$;
(iii) *if* $a \trianglelefteq a'$ *and* $b \trianglelefteq b'$, *then* $a \multimap b \trianglelefteq a' \multimap b'$;
(iv) *if* $f : a \multimap b$ *and* $f' : a' \multimap b'$, *then* $f \multimap f' : (b \multimap a') \multimap (a \multimap b')$;
(v) *if* $f : a \multimap b$ *and* $f' : b \multimap c$, *then* $f' \circ f : a \multimap c$.

Parts (i), (iii), (iv), and (v) can be proved using (4.8) and (4.11) in an algebraic (formal) fashion. It is (ii) that tells us what it all means: the range of $a \multimap b$ consists exactly of those functions which are restricted to (the range of) a and which have values in b. So we can read $u : a \multimap b$ in the normal way: u is a (continuous) mapping from a into b. In technical jargon, we can say that the (strict) retracts and continuous functions form a *category*. In fact, it is equivalent to the category of separable continuous lattices and continuous maps. In this context, (iv) shows that $\multimap$ operates not only on spaces (retracts) but also on maps: it is a *functor* contravariant in the first argument and covariant in the second. Further categorical properties will emerge.

THEOREM 4.4 (The product theorem). *Suppose a, b, a', b' are retracts. Then we have*:

(i) $a \otimes b$ *is a retract, and it is strict if* a *and* b *are*;
(ii) $u : a \otimes b$ *iff* $u = \langle u_0, u_1 \rangle$ *and* $u_0 : a$ *and* $u_1 : b$;
(iii) *if* $a \trianglelefteq a'$ *and* $b \trianglelefteq b'$, *then* $a \otimes b \trianglelefteq a' \otimes b'$;
(iv) *if* $f : a \multimap b$ *and* $f' : a' \multimap b'$, *then* $f \otimes f' : a \otimes a' \multimap b \otimes b'$.

Again the operator proves to be a functor, but what is stated in Theorem 4.4 is not quite enough for the standard identification of $\otimes$ as the categorical product. For this we need some additional combinators:

$$\mathbf{fst} = \lambda u.u_0; \tag{4.14}$$

$$\mathbf{snd} = \lambda u.u_1; \tag{4.15}$$

$$\mathbf{diag} = \lambda u.\langle u, u \rangle. \tag{4.16}$$

Then we have these properties:

$$\mathbf{fst} \circ (a \otimes b) : (a \otimes b) \multimap a; \tag{4.17}$$

$$\mathbf{snd} \circ (a \otimes b) : (a \otimes b) \multimap b; \tag{4.18}$$

$$\mathbf{diag} \circ a : a \multimap a \otimes a; \tag{4.19}$$

$$\mathbf{fst} \circ (f \otimes f') = f \circ \mathbf{fst}; \tag{4.20}$$

$$\mathbf{snd} \circ (f \otimes f') = f' \circ \mathbf{snd}. \tag{4.21}$$

Here a and b are retracts and f and f' are functions. Now suppose a, b, and c are retracts and $f : c \multimap a$ and $g : c \to b$. Let

$$h = (f \otimes g) \circ \mathbf{diag} \circ c.$$

We can readily prove that:

$$h : c \multimap (a \otimes b),$$

and

$$\mathbf{fst} \circ h = f \quad \text{and} \quad \mathbf{snd} \circ h = g.$$

Furthermore, h is the *unique* such function. It is this uniqueness and existence property of functions into $a \otimes b$ that identifies the construct as a product.

There are important connections between $\multimap$ and $\otimes$. To state these we require some additional combinators:

$$\mathbf{eval} = \lambda u.u_0(u_1); \tag{4.22}$$

$$\mathbf{curry} = \lambda u \lambda x \lambda y.u(\langle x, y\rangle). \tag{4.23}$$

If a, b, and c are retracts, the mapping properties are:

$$\mathbf{eval} \circ ((b \multimap c) \otimes b) : ((b \multimap c) \otimes b) \multimap c \tag{4.24}$$

$$\mathbf{curry} \circ ((a \otimes b) \multimap c) : ((a \otimes b) \multimap c) \multimap (a \multimap (b \multimap c)). \tag{4.25}$$

Suppose next that $f : (a \otimes b) \multimap c$ and $g : a \multimap (b \multimap c)$. We find that

$$\mathbf{eval} \circ (\mathbf{curry}(f) \otimes b) = f$$

and

$$\mathbf{curry}(\mathbf{eval} \circ (g \otimes b)) = g.$$

This shows that our category of retracts is a *Cartesian closed category*, which means roughly that product spaces and function spaces within the category interact harmoniously.

THEOREM 4.5 (The sum theorem). *Suppose a, b, a', b' are retracts. Then we have*

(i) *$a \oplus b$ is a retract, and it is always strict;*

(ii) *$u : a \oplus b$ iff $u = \bot$ or $u = \top$ or*

$$u = \langle 0, u_1\rangle \text{ and } u_1 : a \text{ or}$$

$$u = \langle 1, u_1\rangle \text{ and } u_1 : b;$$

(iii) *if $a \Subset a'$ and $b \Subset b'$, then $a \oplus b \Subset a' \oplus b'$;*

(iv) *if $f : a \multimap b$ and $f' : a' \multimap b'$, then $f \oplus f' : a \oplus a' \multimap b \oplus b'$.*

There are several combinators associated with $\oplus$:

$$\mathbf{inleft} = \lambda x.\langle 0, x\rangle; \tag{4.26}$$

$$\mathbf{inright} = \lambda x.\langle 1, x\rangle; \tag{4.27}$$

$$\mathbf{outleft} = \lambda u.u_0 \supset u_1, \mid \bot; \tag{4.28}$$

$$\mathbf{outright} = \lambda u.u_0 \supset \bot, u_1; \tag{4.29}$$

$$\mathbf{which} = \lambda u.u_0; \tag{4.30}$$

$$\mathbf{out} = \lambda u.u_1. \tag{4.31}$$

(The last two are the same as **fst** and **snd**, but they will be used differently.) We find:

$$(a \oplus b) \circ \mathbf{inleft} \circ a : a \rightarrowtail (a \oplus b); \tag{4.32}$$

$$(a \oplus b) \circ \mathbf{inright} \circ b : b \rightarrowtail (a \oplus b); \tag{4.33}$$

$$a \circ \mathbf{outleft} \circ (a \oplus b) : (a \oplus b) \rightarrowtail a; \tag{4.34}$$

$$b \circ \mathbf{outright} \circ (a \oplus b) : (a \oplus b) \rightarrowtail b; \tag{4.35}$$

$$\mathbf{which} \circ (a \oplus b) : (a \oplus b) \rightarrowtail \mathbf{bool}; \tag{4.36}$$

$$a \circ \mathbf{out} \circ (a \oplus a) : (a \oplus a) \rightarrowtail a; \tag{4.37}$$

where a and b are retracts. Most of these facts as they stand are trivial until one sets down the relations between all these maps; but there are too many to put them down here. Note, however, if a, b, and c are retracts and $f : a \rightarrowtail c$ and $g : b \rightarrowtail c$, then if we let

$$h = c \circ \mathbf{out} \circ (f \oplus g),$$

we have:

$$h : (a \oplus b) \rightarrowtail c,$$

and

$$h \circ \mathbf{inleft} = f \quad \text{and} \quad h \circ \mathbf{inright} = g.$$

But, though h exists, it is *not* unique. So $a \oplus b$ is not the categorical sum (coproduct). The author does not know a neat categorical characterization of this operator.

There would be no difficulty in extending $\otimes$ and $\oplus$ to more factors by expanding the range of indices from 0, 1 to $0, 1, \cdots, n-1$. The explicit formulae need not be given; but if we write $a_0 \otimes a_1 \otimes \cdots \otimes a_{n-1}$, we intend this expanded meaning rather than the iterated binary product.

To understand sums and other facts about retracts, consider the least fixed point of this equation:

$$\mathbf{tree} = \mathbf{nil} \oplus (\mathbf{tree} \otimes \mathbf{tree}). \tag{4.38}$$

To be certain that **tree** *is* a retract, we need a general theorem:

THEOREM 4.6 (The limit theorem). *Suppose F is a continuous function that maps retracts to retracts and let $c = \mathbf{Y}(F)$. Then c is also a retract. If in addition F maps strict retracts to strict retracts and is monotone in the sense that $a \ll b$ implies $F(a) \ll F(b)$ for all (strict) retracts a and b, then the range of c is homeomorphic to the inverse limit of the ranges of the strict retracts $F^n(\perp)$ for $n \in \omega$.*

This can be applied in the case of (4.38) where $F = \lambda z.\mathbf{nil} \oplus (z \otimes z)$. Thus we can analyze **tree** as an inverse limit. This approach has the great advantage over the earlier method of the author where limits were required in showing that **tree** *exists*. Here we use **Y** to give existence at once, and then apply Theorems 4.3–4.5 to figure out the nature of the retract.

In Theorem 4.6, the fact that c is a retract can be reasoned as follows: $\perp$ is a

retract. Thus each $F^n(\perp)$ is a retract. We compute:

$$\begin{aligned} c \circ c &= \bigcup\{F^n(\perp) | n \in \omega\} \circ \bigcup\{F^n(\perp) | n \in \omega\} \\ &= \bigcup\{F^n(\perp) \circ F^n(\perp) | n \in \omega\} \qquad (\textit{Note}: \text{same } n.) \\ &= \bigcup\{F^n(\perp) | n \in \omega\} = c. \end{aligned}$$

In case F is monotone and preserves strictness, then we can argue that each $F^n(\perp) \ll c$. The retracts $F^n(\perp)$ are the *projections* of c onto the terms of the limit. Of course $F^n(\perp) \ll F^m(\perp)$ if $n \leqq m$. The $u : c$ can be put into a one-to-one correspondence (homeomorphism, lattice isomorphism) with the infinite sequences $\langle v_0, v_1, \cdots, v_n, \cdots \rangle$, where $v_n : F^n(\perp)$ and $v_n = F^n(\perp)(v_{n+1})$. Indeed $v_n = F^n(\perp)(u)$ and $u = \bigcup\{v_n | n \in \omega\}$. This is exactly the inverse limit construction.

Retreating from generalities back to the example of **tree**, we can grant that it exists and is provably a retract. Two things in its range are $\perp$ and $\top$ by Theorem 4.5(ii), but they are not so interesting. Now $\perp$: **nil**, so by Theorem 4.5(ii) we have $\langle 0 \rangle = \langle 0, \perp \rangle$: **tree**. Let us think of this as *the* atom. What else can we have? If x, y : **tree**, then $\langle x, y \rangle$: **tree** $\otimes$ **tree** and so $\langle 1, \langle x, y \rangle \rangle$: **tree**. Thus (the range of) **tree** contains an atom and is closed under a binary operation. Note that the atomic and nonatomic trees are distinguished by **which** and that suitable constructor and destructor functions are definable on **tree**. But the space also contains infinite trees since we can solve for the least fixed point of:

$$t = \langle 1, \langle\langle 0 \rangle, t \rangle\rangle$$

and t : **tree**. (Why?) And there are many other examples of infinite elements in **tree**.

A point to stress in this construction is that **tree** being LAMBDA-definable is *computable*, and there are many computable functions definable on or to (the range of) **tree**. All the "structural" functions, for example, are computable. These are functions which in other languages would be called **isatom** or **construct** or **node**, and they are all easily LAMBDA-definable. Just as with $\oplus$, $\otimes$, $\multimap$, they are not explicit in the *notation*, but they are definable nevertheless. In the case of **node**, we could use finite sequences of Boolean values to pick out or name nodes. Thus solve for **name** = **nil** $\oplus$ **bool** $\otimes$ **name**, and then give a recursive definition of:

$$\textbf{node} : \textbf{name} \multimap (\textbf{tree} \multimap \textbf{tree}).$$

Any combination of retract preserving functors can be used in this game. For example:

$$\textbf{lamb} = \textbf{int} \oplus (\textbf{lamb} \multimap \textbf{lamb}). \tag{4.39}$$

This looks innocent, but the range of **lamb** would give a quite different and not unattractive model for the λ-calculus (plus arithmetic). What we do to investigate this model is to modify LAMBDA slightly by replacing the ternary conditional $z \supset x, y$ by a *quarternary* one $w \supset x, y, z$; otherwise the syntax of the language remains the same. The semantics, however, is a little more complex.

Let us use τ, σ, ρ, θ as syntactical variables for expressions in the modified language. The semantics is provided by a function $\mathscr{K}$ that maps the expressions of the language to their values in (the range of) **lamb**. To be completely rigorous we

also have to confront the question of free and bound variables. For simplicity let us index the variables of the language by integers, and let us take the variables to be $v_0, v_1, v_2, \cdots, v_n, \cdots$. We cannot simply evaluate out an expression τ to its value $\mathscr{H}[\![\tau]\!]$ until we know the values of the free variables in τ. The values of these variables will be given by an "environment" t which can be construed as a *sequence* of values in **lamb**. We can restrict these environments to the retract:

$$\textbf{env} = \lambda t.\textbf{lamb} \circ \textbf{seq}(t). \tag{4.40}$$

When t : **env**, then t_n : **lamb** is the value that the environment gives to the variable v_n. We also need to employ a transformation on environments as follows:

$$t[x/n] = \lambda m \in \omega.\textbf{eq}(n)(m) \supset x, t_m. \tag{4.41}$$

Here **eq** is the primitive recursive function that is 0, if n, m are equal, and is 1, otherwise, for $n, m \in \omega$. The effect of $t[x/n]$ is to replace the nth term of the sequence t by the value of x, otherwise to leave the rest of the sequence unchanged. To correspond with our use of very simple variables we have selected a simple notion of environment: in the semantics of more general languages it is customary to regard an environment as a function from the set of variables into the domain of denotable values.

The correct way to evaluate a term τ given an environment t is to find $\mathscr{H}[\![\tau]\!](t)$. We use the brackets [\![and]\!] here simply as an aid to the eye in keeping the syntactical part separated from the rest. The environment enters as a function-argument in the usual way; thus we shall have:

$$\mathscr{H}[\![\tau]\!] : \textbf{env} \to\!\!\!\!\circ\ \textbf{lamb}. \tag{4.42}$$

$$\begin{aligned}
&\mathscr{H}[\![v_n]\!](t) = t_n \\
&\mathscr{H}[\![0]\!](t) = \textbf{inleft}(0) \\
&\mathscr{H}[\![\tau+1]\!](t) = \textbf{which}(\mathscr{H}[\![\tau]\!](t)) \supset \textbf{inleft}(\textbf{out}(\mathscr{H}[\![\tau]\!](t))+1),\ \perp \\
&\mathscr{H}[\![\tau-1]\!](t) = \textbf{which}(\mathscr{H}[\![\tau]\!](t)) \supset \textbf{inleft}(\textbf{out}(\mathscr{H}[\![\tau]\!](t))-1),\ \perp \\
&\mathscr{H}[\![\theta \supset \tau, \sigma, \rho]\!](t) = \textbf{lamb}(\textbf{which}(\mathscr{H}[\![\theta]\!](t)) \supset \\
&\qquad\qquad (\textbf{out}(\mathscr{H}[\![\theta]\!](t)) \supset \mathscr{H}[\![\tau]\!](t), \mathscr{H}[\![\sigma]\!](t)), \mathscr{H}[\![\rho]\!](t)) \\
&\mathscr{H}[\![\tau(\sigma)]\!](t) = \textbf{which}(\mathscr{H}[\![\tau]\!](t)) \supset \perp, \textbf{out}(\mathscr{H}[\![\tau]\!](t))(\mathscr{H}[\![\sigma]\!](t)) \\
&\mathscr{H}[\![\lambda v_n.\tau]\!](t) = \textbf{inright}(\lambda x : \textbf{lamb}.\mathscr{H}[\![\tau]\!](t[x/n])).
\end{aligned} \tag{4.43}$$

A good question is: why does $\mathscr{H}$ exist? The answer is: because of the fixed-point theorem.

If we rewrite the semantic equations $\mathscr{H}[\![\tau]\!](t) = (\cdots)$ in (43) by the equation $\mathscr{H}[\![\tau]\!] = \lambda t : \textbf{env}(\cdots)$, then $\mathscr{H}$ is seen to be a function from expressions to values in **lamb**. As the range of **lamb** is contained in $\mathbf{P}\omega$, we can say more broadly that $\mathscr{H} \in \mathbf{P}\omega^{\mathbf{Exp}}$, where **Exp** is the syntactical set of expressions and the exponential notation designates the set of *all* functions from **Exp** into $\mathbf{P}\omega$. This function set is a complete lattice because $\mathbf{P}\omega$ is. Therefore if we read (4.43) as a definition by cases on **Exp**, then we can find $\mathscr{H}$ as a suitable fixed point in the complete lattice $\mathbf{P}\omega^{\mathbf{Exp}}$. Indeed it is the fixed point of a continuous operator.

Actually we can regard **Exp** as being a *subset* of $\mathbf{P}\omega$ to avoid dragging in other lattices. What we need is another recursive definition of a data type:

$$\begin{aligned}\mathbf{exp} = \mathbf{int} \oplus \mathbf{nil} \oplus \mathbf{exp} \oplus \mathbf{exp} \oplus (\mathbf{exp} \otimes \mathbf{exp} \otimes \mathbf{exp} \otimes \mathbf{exp}) \\ \oplus (\mathbf{exp} \otimes \mathbf{exp}) \oplus (\mathbf{int} \otimes \mathbf{exp})\end{aligned} \tag{4.44}$$

Note that there are as many summands in (4.44) as there are clauses in (4.43). We can think of **exp** as giving the "abstract" syntax of the language. We use the integers to index the variables and the nil element to stand for the individual constant. Read (4.44) as saying that every expression is *either* a variable *or* a constant *or* the successor of an expression *or* the predecessor of an expression *or* the conditional formed from a tuple of expressions *or* the abstraction formed from a pair of a variable and an expression. We do not need in (4.44) to introduce special "symbols" for the successor, application, etc., because the separation by cases given by the $\oplus$ operation is sufficient to make the distinctions. (That is why the syntax is "abstract".) The point is that for recursive definitions it does not matter how we make the distinctions as long as they can be made. From this new point of view, we could rewrite (4.43) so as to show:

$$\mathscr{H} : \mathbf{exp} \multimap (\mathbf{env} \multimap \mathbf{lamb}), \tag{4.45}$$

which is clearly more satisfactory—especially as it is now clear that $\mathscr{H}$ is *computable.* And this is a method that can be generalized to many other languages. The method also shows why it is useful to allow *function spaces* as particular data types.

Another example of this method can be illustrated, if the reader will recall the Gödel numbering of § 3. It will be seen that there are similarities with the **tree** construction: instead of 0 and $\mathbf{apply}(n)(m)$, **tree** uses $\langle 0 \rangle$ and $\langle 1, \langle x, y \rangle \rangle$. Note, however, that Gödel numbers are *finite* while **tree** has *infinite* objects. But the infinite objects are always *limits* of finite objects, so there *are* connections. (We discuss this again in § 6.) In particular, recursive definitions on Gödel numbers, like that of **val**, have analogues on **tree**. Here is the companion of (3.7).

$$\mathbf{vaal} = \lambda x : \mathbf{tree}.\mathbf{which}(x) \supseteq \mathbf{G}, \mathbf{vaal}(\mathbf{fst}(\mathbf{out}(x)))(\mathbf{vaal}(\mathbf{snd}(\mathbf{out}(x)))). \tag{4.46}$$

We have $\mathbf{vaal} : \mathbf{tree} \multimap \mathbf{id}$, where of course (4.46) is taken as defining **vaal** as the least fixed point. This is an example of a computable function between effectively given retracts. The LAMBDA-definable elements of $\mathbf{P}\omega$ are the computable elements in the range of **vaal**.

We have discussed the category of retracts and continuous maps, but if they are all LAMBDA-definable, then they fall within the countable model **RE**. Thus there is *another* category of effectively given retracts and effectively given continuous maps. (Examples: **tree**, **id**, **vaal**, and all those retracts and maps generated by $\oplus$, $\otimes$, and $\multimap$.) This category seems to deserve the status of a generalized recursion theory; though this is not to say that as yet very much is known about it. In fact, the proper formulation may require an enriched category rather than a restricted one. Thus instead of confining attention to the computable retracts and computable maps, it might be better to use the full category with all maps and to single out the computable ones (also maybe the finite ones) by special predicates. In effect we have avoided any methodological decisions by working in

the universal space $\mathbf{P}\omega$ and by *defining* a notion when required—if possible with the aid of LAMBDA. This makes it possible to give all the necessary definitions and to prove the theorems without at first having to worry about axiomatic problems.

5. Closure operations and algebraic lattices. Given any family of (finitary) operations on a set (say, ω) there is a closure operation defined on the subsets of that set obtained by forming the *least subset* including the given elements and *closed* under the operations. Examples are very familiar from algebra: the subgroup generated by a set of elements, the subspace spanned by a set of vectors, the convex hull of a set of geometric points. We simplify matters here by restricting attention to closures operating on sets in $\mathbf{P}\omega$, but the idea is quite general. The main point about these "algebraic" closure operations—as distinguished from topological closure operations—is that they are *continuous*. Thus, in the case of subgroups, if an element belongs to the subgroup generated by some elements, then it also belongs to the subgroup generated by *finitely* many of them. In the context of $\mathbf{P}\omega$ we can state the characteristic condition very simply.

DEFINITION. An element $a \in \mathbf{P}\omega$ is called a *closure operation* iff it satisfies: $\mathbf{I} \subseteq a = a \circ a$.

We see by definition that a closure operation is not only continuous, but it is also a retract. This is reasonable since the closure of the closure of a subset must be equal to the closure. To say of a function that $\mathbf{I} \subseteq a$, means that $x \subseteq a(x)$ for all $x \in \mathbf{P}\omega$. In other words, every set is contained in its closure. (Note that closures are opposite to the "projections", those retracts where $a \subseteq \mathbf{I}$.) Among examples of closure operations we find $\mathbf{I}$ and $\top$; the first has the most closed sets (fixed points), the second has the least. (Note that $\top = \omega$ always is a fixed point of a closure operation; $\top = \lambda x.\top$ is thus the most trivial closure operation.) The examples **fun**, **open**, **int** of § 4 are all closure operations (cf. (4.1), (4.6), (4.7)). We remarked that **fun** is a retract, but the reader should prove in addition:

$$u \subseteq \lambda x.u(x), \tag{5.1}$$

for all $u \in \mathbf{P}\omega$ (cf. Theorem 1.2). We note that this fact can be rewritten in the language of retracts as:

$$\mathbf{I} \subseteq \mathbf{I} \rightarrowtail \mathbf{I}, \tag{5.2}$$

the significance of which will emerge after we develop a bit of the theory of closure operations.

Unfortunately the natural definition of the retract **bool** does not yield a closure operation. In this section we adopt this modification:

$$\mathbf{boool} = \lambda u.u \supseteq 0, \top + 1. \tag{5.3}$$

The closed sets of **boool** are $\bot$, 0, $\top+1$, and $\top$. Note that with any closure operation a, the function value $a(x)$ is the least closed set (fixed point of a) including as a subset the given set x. Thus given any family $\mathscr{C} \subseteq \mathbf{P}\omega$ of "closed" sets which is closed under the intersection of subfamilies, if we define

$$a(x) = \bigcap\{y \in \mathscr{C} \mid x \subseteq y\}, \tag{5.4}$$

then this will be a closure operation *provided* it is continuous. This remark makes it easy to check that certain functions are closure operations if we can spot easily the family $\mathscr{C}$ of fixed points.

Alas, the "natural" definition of ordered pairs (cf. (2.21)) leads to projections rather than closures. Here we must choose another:

$$[x, y] = \{2n | n \in x\} \cup \{2m+1 | m \in y\}, \tag{5.5}$$

with these inverse functions:

$$[u]_0 = \{n | 2n \in u\}, \tag{5.6}$$

$$[u]_1 = \{m | 2m+1 \in u\}. \tag{5.7}$$

We shall find that the main advantage of these equations lies in the obvious equation:

$$u = [[u]_0, [u]_1], \tag{5.8}$$

which is not true for the other pairing functions. Of course we have:

$$[[x, y]]_0 = x, \tag{5.9}$$

$$[[x, y]]_1 = y. \tag{5.10}$$

We shall not extend the idea of these new functions to triples and sequences, though it is clear what to do.

Abstractly, an *algebraic lattice* is a complete lattice in which the isolated points are dense. An isolated (sometimes called: *compact*) point in a lattice is one that is not the limit (sup or l.u.b.) of any directed family of its proper subelements. This definition works in continuous lattices, but more generally it is better to say that if the isolated point is *contained in* a sup, then it is also contained in a finite subsup (a sup of a finite selection of elements out of the given sup). In the case of the lattice of subgroups of a group, the isolated ones are the finitely generated subgroups. The isolated points of $\mathbf{P}\omega$ are the finite sets e_n. To say that isolated points are *dense* means that every element in the lattice is the sup of the isolated points it contains. The sequel to Theorem 4.1 for closure operations relates them to algebraic lattices.

THEOREM 5.1 (The algebraic lattice theorem). *The fixed points of any closure operation form an algebraic lattice.*

The proof is very easy if one notes that the isolated points of $\{x | x = a(x)\}$, where a is a closure operation, are exactly the images $a(e_n)$ of the finite sets in $\mathbf{P}\omega$. What makes Theorem 5.1 more interesting is the converse.

THEOREM 5.2 (The representation theorem for algebraic lattices). *Every algebraic lattice with a countable number of isolated points is isomorphic to the range of some closure operation.*

By Theorem 1.6 we know that the algebraic lattice is a retract, but a more direct argument makes the closure property clear. Thus, let D be the algebraic lattice with $\{d_n | n \in \omega\}$ as the set of all isolated points with the indicated enumeration. We shall use the square notation with symbols $\sqsubseteq$ and $\bigsqcup$ for the lattice ordering and sup. The desired closure operation is defined by:

$$a(x) = \{m | d_m \sqsubseteq \bigsqcup \{d_n | n \in x\}\}.$$

It is an easy exercise to show that from the definition of "isolated" it follows that a is continuous; and from density, it follows that D is in a one-to-one order preserving correspondence with the fixed points of a.

In the last section we introduced an algebra of retracts, much of which carries over to closure operations given the proper definitions. Without any change we can use Theorem 4.3 on function spaces, provided we check that the required retracts are closures.

THEOREM 5.3 (The function space theorem for algebraic lattices). *Suppose that a and b are closure operations; then so is $a \rightarrow b$.*

The proof comes down to showing that:

$$u(x) \subseteq b(u(a(x))), \tag{5.11}$$

whenever a and b are closure operations. But this is easy by monotonicity. Note that (5.1) is needed.

For those interested in topology, one can give a construction of the isolated points of the function space which is much more direct than just taking the functions $b \circ e_n \circ a$, which on the face of it do not tell us too much. But we shall not need this explicit construction here.

The reason for changing the pairing functions is to be able to form products and sums of closure operations. In the case of products, the analogue of $\otimes$ is straightforward:

$$a \boxtimes b = \lambda u.[a([u]_0), b([u]_1)]; \tag{5.12}$$

while for sums using $a' = \lambda x.0 \cup a(x-1)+1$ and similarly for b' we write:

$$a \boxplus b = \lambda u.([u_0]_0 \supset 0, 0) \cup ([u]_1 \supset 1, 1) \ni [a'([u]_0), \bot], [\bot, b'([u]_1)]. \tag{5.13}$$

We can then establish with the aid of (5.8)–(5.10):

THEOREM 5.4 (The product and sum theorem for algebraic lattices). *Suppose that a and b are closure operations; then so are $a \boxtimes b$ and $a \boxplus b$. Analogues of the results in Theorems* 4.4 *and* 4.5 *carry over.*

Following the discussion in § 4, we can also show that the closure operations form a Cartesian closed category, which in some ways is better than the category of *all* retracts. What makes it better is the existence of a "universe".

Every continuous operation generates a closure operation by just closing up the sets under the continuous function (as a set operation). We can institutionalize this thought by means of this definition:

$$\mathbf{V} = \lambda a \lambda x. \mathbf{Y}(\lambda y.x \cup a(y)). \tag{5.14}$$

Clearly $\mathbf{V}$ is LAMBDA-definable, continuous, etc. A more understandable characterization would define $\mathbf{V}(a)(x)$ by this equation:

$$\mathbf{V}(a)(x) = \bigcap\{y | x \subseteq y \text{ and } a(y) \subseteq y\}. \tag{5.15}$$

These two definitions are easily seen to be equivalent. What is unexpected is the discovery (due in a different form to Peter Hancock and Per Martin-Löf) that $\mathbf{V}$ itself is a closure operation.

THEOREM 5.5 (The universe theorem for algebraic lattices). *The function $\mathbf{V}$ is a closure operation and its fixed points comprise the set of all closure operations.*

Thus to say a is a closure operation, write $a : \mathbf{V}$. To have a mapping on closure operations, write $f : \mathbf{V} \to \mathbf{V}$. Remark that 5.5 allows us to write $\mathbf{V} : \mathbf{V}$. It all seems rather circular, but it is quite consistent. The category of separate algebraic lattices "contains itself"—if we are careful to work through retracts of $\mathbf{P}\omega$.

The proof of Theorem 5.5 requires a few steps. We note first that for all $x, a \in \mathbf{P}\omega$:

$$x \subseteq \mathbf{V}(a)(x). \tag{5.16}$$

Let $y = \mathbf{V}(a)(x)$. This is the least y with $x \cup a(y) \subseteq y$. What is the least z with $y \cup a(z) \subseteq z$? The answer is of course y, which proves:

$$\mathbf{V}(a)(\mathbf{V}(a)(x)) = \mathbf{V}(a)(x). \tag{5.17}$$

Thus $\mathbf{V}(a)$ is always a closure operation. If a is already a closure operation, then clearly $\mathbf{V}(a)(x) = a(x)$. Therefore we have shown:

$$a = \mathbf{V}(a) \quad \text{iff} \quad a \text{ is a closure operation.} \tag{5.18}$$

But then by (5.16) and (5.17) we have by (5.18):

$$\mathbf{V}(a) = \mathbf{V}(\mathbf{V}(a)). \tag{5.19}$$

From (5.16) by monotonicity we see:

$$a(x) \subseteq a(\mathbf{V}(a)(x)) \subseteq \mathbf{V}(a)(x). \tag{5.20}$$

Hence by (5.1) we can derive:

$$a \subseteq \lambda x.a(x) \subseteq \lambda x.\mathbf{V}(a)(x) = \mathbf{V}(a). \tag{5.21}$$

From (5.19) and (5.21) it follows that $\mathbf{V}$ itself is a closure operation.

The operation $\mathbf{V}$ forms the least closure operation containing a given element, and it shows that the lattice of closure operations is not only a retract of $\mathbf{P}\omega$ but also an algebraic lattice. Since we can now use $\mathbf{V}$ as a retract, the earlier results become formulas:

$$(\lambda a : \mathbf{V}.\lambda b : \mathbf{V}.a \boxtimes b) : \mathbf{V} \to (\mathbf{V} \to \mathbf{V}); \tag{5.22}$$

$$(\lambda a : \mathbf{V}.\lambda b : \mathbf{V}.a \boxplus b) : \mathbf{V} \to (\mathbf{V} \to \mathbf{V}); \tag{5.23}$$

we can also state such functorial properties as:

$$(\lambda a : \mathbf{V}.\lambda b : \mathbf{V}.a \to b) : \mathbf{V} \to (\mathbf{V} \to \mathbf{V}). \tag{5.24}$$

Using this style of notation we have:

THEOREM 5.6 (The limit theorem for algebraic lattices).

$$(\lambda f : \mathbf{V} \to \mathbf{V}.\mathbf{Y}(f)) : (\mathbf{V} \to \mathbf{V}) \to \mathbf{V}.$$

In words: if f is a mapping on closure operations, then its least fixed point is also a closure operation. The proof of course holds with *any* retract in place of $\mathbf{V}$, but we are more interested in applications to $\mathbf{V}$. For example, note that $\mathbf{V}(\perp) = \mathbf{I}$. Now let $f = \lambda a : \mathbf{V}.a \to a$. The least fixed point of this f is the limit of the sequence:

$$\perp, \quad \mathbf{I}, \quad \mathbf{I} \to \mathbf{I}, \quad (\mathbf{I} \to \mathbf{I}) \to (\mathbf{I} \to \mathbf{I}), \quad ((\mathbf{I} \to \mathbf{I}) \to (\mathbf{I} \to \mathbf{I})) \to ((\mathbf{I} \to \mathbf{I}) \to (\mathbf{I} \to \mathbf{I})), \cdots,$$

and we see that all these retracts are *strict*. This means $\mathbf{Y}(f)$ is nontrivial in that it has at least *two* fixed points (viz., $\perp$ and $\top$). But $d = \mathbf{Y}(f)$ must be the least closure operation satisfying

$$d = d \multimap d, \tag{5.25}$$

and we have thus proved that there are *nontrivial* algebraic lattices isomorphic to their own function spaces. This construction (which rests on hardly more than (5.2), since we could take $d = \mathbf{Y}(\lambda a.\mathbf{I} \cup (a \multimap a))$) is much quicker than the inverse limit construction originally found by the author to give λ-calculus models satisfying (η). There are many other fixed points of (5.25) besides this least closure operation, but their connection with inverse limits is not fully investigated.

We note in conclusion that most constructions by fixed points give algebraic lattices (like **lamb** in § 4), and so we could just as well do them in **V** if we remember to use $\boxtimes$ and $\boxplus$. The one-point space is $\top$ (*not* **nil**), and so the connection with inverse limits via Theorem 4.6 is not as clear when nonstrict functions are used. For many purposes, this may not make any difference.

6. Subsets and their classification. Retracts produce very special subsets of $\mathbf{P}\omega$: a retract always has a nonempty range which forms a lattice under $\subseteq$. For example, the range of **int** is $\{\perp, \top\} \cup \omega$. We often wish to eliminate $\perp$ and $\top$; and with a retract like **tree** the situation is more complex, since combinations like $\langle 1, \langle\langle 1, \langle \perp, \langle 0\rangle\rangle\rangle, \top\rangle\rangle$ might require elimination. In these two cases the method is simple.

Consider these two functions:

$$\mathbf{mid} = \lambda x : \mathbf{int}.\ x \supset 0, 0 \tag{6.1}$$

$$\mathbf{perf} = \lambda u : \mathbf{tree}.\mathbf{which}(u) \supset 0, \Delta(\mathbf{perf}(\mathbf{fst}(\mathbf{out}(u))))(\mathbf{perf}(\mathbf{snd}(\mathbf{out}(u)))) \tag{6.2}$$

where Δ is a special combinator:

$$\Delta = \lambda x \lambda y.(x \supset (y \supset 0, \top), \top) \cup (y \supset (x \supset 0, \top), \top). \tag{6.3}$$

We find that $\omega = \{x : \mathbf{int} | \mathbf{mid}(x) = 0\}$. In the case of trees, note first this behavior of Δ:

Δ	$\perp$	0	$\top$
$\perp$	$\perp$	$\perp$	$\top$
0	$\perp$	0	$\top$
$\top$	$\top$	$\top$	$\top$

The question is: what subset is $\{u : \mathbf{tree} | \mathbf{perf}(u) = 0\}$?

Now **perf** is defined recursively. We can see that

$$\mathbf{perf}(\perp) = \perp, \quad \mathbf{perf}(\top) = \top, \quad \mathbf{perf}(\langle 0\rangle) = 0,$$

and

$$\mathbf{perf}(\langle 1, \langle x, y\rangle\rangle) = \Delta(\mathbf{perf}(x))(\mathbf{perf}(y))$$

when $x, y : \mathbf{tree}$. Every tree, aside from $\top$ or $\perp$, is either atomic or a pair of trees. The atomic tree is "perfect" (that is, $\mathbf{perf}(\langle 0\rangle) = 0$). A *finite* tree which does not

contain $\bot$ or $\top$ is perfect—as we can see inductively using the table above for Δ. An infinite tree is never perfect: either some branch ends in $\top$ and **perf** maps it to $\top$, or $\top$ is never reached and **perf** maps it to $\bot$. Thus the subset in question is then seen to be the set of *finite* trees generated from the atom by pairing. This is clearly a desirable subset, and it is sorted out by a function with a simple recursive definition. The general question is: what subsets can be characterized by equations? The answer can be given by reference to the *topology* of $\mathbf{P}\omega$.

DEFINITION. Let $\mathfrak{G}$ be the class of *open* subsets of $\mathbf{P}\omega$, and $\mathfrak{F}$ be the class of *closed* subsets. Further let $\mathfrak{B}$ be the class of all (finite) *Boolean combinations* of open sets.

We recall from § 1 that $U \in \mathfrak{G}$ just in case for all $x \in \mathbf{P}\omega$, we have $x \in U$ if and only if some finite subset of x is in U. The class of open sets contains $\varnothing$ and $\mathbf{P}\omega$ and is closed under finite intersection and arbitrary union; in fact, it can be generated by these two closure conditions from subsets of the special form $\{x \in \mathbf{P}\omega \mid n \in x\}$ for the various $n \in \omega$ An open set is always *monotonic* (whenever $x \in U$ and $x \subseteq y$, then $y \in U$), so that every nonempty $U \in \mathfrak{G}$ has $\top \in U$.

Another characterization of openness can be given by continuous functions. Suppose $U \in \mathfrak{G}$. Define $f : \mathbf{P}\omega \to \{\bot, \top\}$ so that

$$U = \{x \mid f(x) = \top\};$$

then f is continuous. Conversely, if such an f is continuous, then U is open. But if we do not assume the range of f is included in $\{\bot, \top\}$, this is not true. For the case of general functions we know that f is continuous if and only if $\{x \mid f(x) \in V\}$ is open for all open V. This defines continuity in terms of openness, but we can turn it the other way around:

THEOREM 6.1 (The $\mathfrak{G}$ theorem). *The open subsets of* $\mathbf{P}\omega$ *are exactly the sets of the form*:

$$\{x \mid f(x) \supseteq 0\},$$

where $f : \mathbf{P}\omega \to \mathbf{P}\omega$ *is continuous.*

We could have written $0 \in f(x)$ or the equation $f(x) \cap 0 = 0$ instead of $f(x) \supseteq 0$. Note that in case $f : \mathbf{P}\omega \to \{\bot, \top\}$, then $f(x) \supseteq 0$ is equivalent to $f(x) = \top$. Also any other integer could have been used in place of 0.

We can say that $\{x \mid 0 \in x\}$ is the *typical* open set, and that every other open set can be obtained as an inverse image of the typical set by a continuous function. We shall extend this pattern to other classes, especially looking for equations. In the case of openness an inequality could also be used, giving as the typical set $\{x \mid x \neq \bot\}$. But since closed sets are just the complements of open sets, this remark gives us:

THEOREM 6.2 (The $\mathfrak{F}$ theorem). *The closed subsets of* $\mathbf{P}\omega$ *are exactly the sets of the form*:

$$\{x \mid f(x) = \bot\},$$

where $f : \mathbf{P}\omega \to \mathbf{P}\omega$ *is continuous.*

Aside from $\{x \mid x = \bot\}$, we could have used $\{x \mid x \subseteq a\}$ as the typical closed set where $a \in \mathbf{P}\omega$ is any element whatsoever aside from $\top$. This $\top$ has, by the way, a

special character. We note:

$$\{\top\} = \bigcap\{\{x | n \in x\} | n \in \omega\}.$$

Thus $\{\top\}$ is a *countable intersection* of open sets, otherwise called a $\mathfrak{G}_\delta$-set. There are of course many other $\mathfrak{G}_\delta$-sets, but $\{\top\}$ is the typical one:

THEOREM 6.3 (The $\mathfrak{G}_\delta$ theorem). *The countable intersections of open subsets of* $\mathbf{P}\omega$ *are exactly the sets of the form*:

$$\{x | f(x) = \top\},$$

where $f: \mathbf{P}\omega \to \mathbf{P}\omega$ *is continuous.*

It may not be obvious that every $\mathfrak{G}_\delta$-set has this form. Certainly, as we have remarked, every $\mathfrak{G}$-set has this form. Thus if W is a $\mathfrak{G}_\delta$, we have:

$$W = \bigcap\{U_n | n \in \omega\}$$

and further,

$$U_n = \{x | f_n(x) = \top\},$$

where the f_n are suitably chosen continuous functions. Define the function g by the equation:

$$g(x) = \{(n, m) | m \in f_n(x)\}.$$

Clearly g is continuous, and we have:

$$W = \{x | g(x) = \top\},$$

as desired.

We let $\mathfrak{F} \dot{\cap} \mathfrak{G}$ denote the class of all sets of the form $C \cap U$, where $C \in \mathfrak{F}$ and $U \in \mathfrak{G}$. Similarly for $\mathfrak{F} \dot{\cap} \mathfrak{G}_\delta$. Now $\{x | x \subseteq 0\}$ is closed and $\{x | x \supseteq 0\}$ is open. Thus $\{0\} \in \mathfrak{F} \dot{\cap} \mathfrak{G}$. This set is typical.

THEOREM 6.4 (The $\mathfrak{F} \dot{\cap} \mathfrak{G}$ theorem). *The sets that are intersections of closed sets with open sets are exactly the sets of the form*:

$$\{x | f(x) = 0\},$$

where $f: \mathbf{P}\omega \to \mathbf{P}\omega$ *is continuous.*

Again it may not be obvious that every $\mathfrak{F} \dot{\cap} \mathfrak{G}$ set has this form. We can write:

$$C = \{x | f(x) = \bot\},$$

and

$$U = \{x | g(x) = 0\},$$

where $C \in \mathfrak{F}$ and $U \in \mathfrak{G}$ and the continuous f and g are suitably chosen. Define

$$h(x) = \{2n + 1 | n \in f(x)\} \cup \{2n | n \in g(x)\},$$

and remark that h is continuous. We have:

$$C \cap U = \{x | h(x) = 0\},$$

as desired.

It is easy to see that $\{e\} \in \mathfrak{F} \dot{\cap} \mathfrak{G}$ if e is finite, but in general $\{a\} \in \mathfrak{F} \dot{\cap} \mathfrak{G}_\delta$. In

case a is infinite but not equal to $\top$ (say, $a = \{n | n > 0\} = \top + 1$), then $\{a\}$ is typical in its class.

THEOREM 6.5 (The $\mathfrak{F} \dot{\cap} \mathfrak{G}_\delta$ theorem). *The sets that are intersections of closed sets with countable intersections of open sets are exactly the sets of the form*:

$$\{x | f(x) = a\},$$

where $f : \mathbf{P}\omega \to \mathbf{P}\omega$ *is continuous and* a *is a fixed infinite set not equal to* $\top$.

Note that $(\mathfrak{F} \dot{\cap} \mathfrak{G})_\delta$ is the same class as $\mathfrak{F} \dot{\cap} \mathfrak{G}_\delta$, so we see by Theorem 6.4 that a good choice of a is $\lambda n \in \omega.0$.

There is no single subset of $\mathbf{P}\omega$ typical for $\mathfrak{B}$, which can be viewed as the *finite unions* of sets from the class $\mathfrak{F} \dot{\cap} \mathfrak{G}$.

THEOREM 6.6 (The $\mathfrak{B}$ theorem). *The sets that are Boolean combinations of open sets are exactly the sets of the form*:

$$\{x | f(x) \in \mathscr{E}\},$$

where $f : \mathbf{P}\omega \to \mathbf{P}\omega$ *is continuous and* $\mathscr{E}$ *is a finite set of finite elements of* $\mathbf{P}\omega$.

To see that every $\mathfrak{B}$ set has this form, suppose that

$$V = W_0 \cup W_1 \cup \cdots \cup W_{n-1},$$

where each $W_i \in \mathfrak{F} \dot{\cap} \mathfrak{G}$. We can write:

$$W_i = \{x | f_i(x) = 0\},$$

where $f_i : \mathbf{P}\omega \to \{\bot, 0, \top\}$ is continuous. Then define:

$$g(x) = \{2i + j | j \in f_i(x) \cap \{0, 1\}, i < n\},$$

and note that g is continuous. Let

$$\mathscr{E} = \{y \subseteq \{m | m < 2n\} | \exists i < n. 2i \in y, 2i + 1 \notin y\};$$

we have:

$$V = \{x | g(x) \in \mathscr{E}\}$$

as desired.

THEOREM 6.7 (The $\mathfrak{B}_\delta$ theorem). *The sets that are countable intersections of Boolean combinations of open sets are exactly the sets of the form*:

$$\{x | f(x) = g(x)\},$$

where $f, g : \mathbf{P}\omega \to \mathbf{P}\omega$ *are continuous.*

This is clearly the most interesting of these characterization theorems, because equations like $f(x) = g(x)$ turn up all the time and the collection is a very rich totality of subsets of $\mathbf{P}\omega$. It includes all the retracts, since they are of the form $\{x | x = a(x)\}$. And much more. That every such set in Theorem 6.7 is $\mathfrak{B}_\delta$ follows from these logical transformations:

$$\{x | f(x) = g(x)\} = \{x | \forall n \in \omega [n \in f(x) \leftrightarrow n \in g(x)]\}$$

$$= \bigcap_{n=0}^{\infty} (\{x | n \in f(x), n \in g(x)\} \cup \{x | n \notin f(x), n \notin g(x)\})$$

That puts the set in the class $(\mathfrak{F} \dot{\cup} \mathfrak{G})_\delta \subseteq \mathfrak{B}_\delta$.

On the other hand, we can see that $(\mathfrak{F} \dot{\cup} \mathfrak{G})_\sigma$ is exactly $\mathfrak{B}_\delta$. Because, in view of Theorem 6.6, $\mathfrak{B}_\sigma$, the class of *countable* unions of $\mathfrak{B}$-sets, is exactly $(\mathfrak{F} \dot{\cap} \mathfrak{G})_\sigma$. The remark we want to make then follows by taking complements.

Now let S be an arbitrary $\mathfrak{B}_\delta$-set. We can write:

$$S = \bigcap_{n=0}^{\infty} (\{x | f_n(x) = 0\} \cup \{x | g_n(x) = 0\}),$$

where $f_n, g_n : \mathbf{P}\omega \to \{\bot, 0, \top\}$ are continuous. Now let u, v be continuous functions which on $\{\bot, 0, \top\}$ realize these two tables:

u	$\bot$	0	$\top$
$\bot$	$\bot$	0	0
0	0	0	$0'$
$\top$	0	$0'$	$0'$

v	$\bot$	0	$\top$
$\bot$	0	0	$0'$
0	0	0	$0'$
$\top$	$0'$	$0'$	$\top$

where $0' = 0 \cup 1$. This is an exercise in many-valued logic, and we find for $x, y \in \{\bot, 0, \top\}$:

$$u(x)(y) = v(x)(y) \quad \text{iff} \quad x = 0 \text{ or } y = 0.$$

Thus define continuous functions f' and g' such

$$f' = \lambda x \lambda n \in \omega . u(f_n(x))(g_n(x)),$$

$$g' = \lambda x \lambda n \in \omega . v(f_n(x))(g_n(x)),$$

and we find:

$$S = \{x | f'(x) = g'(x)\}$$

as desired.

This is as far as we can go with *equations*. More complicated sets can be defined using *quantifiers*, for example the Σ^1_1 or *analytic* sets can be put in the form:

$$\{x | \exists y . f(x)(y) = g(x)(y)\},$$

and their complements, the Π^1_1 sets, in the form

$$\{x | \forall y \; \exists z . h(x)(y)(z) = 0\},$$

with continuous f, g, h. For the three classes we then have as "typical" sets those shown in Table 3.

It should be remarked that $\mathfrak{B}_\delta$ contains all the closed sets in the *Cantor space* topology on $\mathbf{P}\omega$ (that is, the topology obtained when it is regarded as the infinite product of *discrete* two-point spaces). Therefore the Σ^1_1 sets for the two topologies on $\mathbf{P}\omega$ are the *same*. Hence, since we know for Cantor space that $\Delta^1_1 = \Sigma^1_1 \cap \Pi^1_1$ is the class of *Borel sets*, we can conclude that the two topologies on $\mathbf{P}\omega$ have the *same* Borel sets. (That is, in both cases Δ^1_1 is the Boolean σ-algebra generated from the open sets.)

TABLE 3
Classes and typical sets

Classes	Typical sets
$\mathfrak{G}$	$\{x \mid 0 \in x\}$
$\mathfrak{F}$	$\{\bot\}$
$\mathfrak{G}_\delta$	$\{\top\}$
$\mathfrak{F} \hat{\cap} \mathfrak{G}$	$\{0\}$
$\mathfrak{F} \hat{\cap} \mathfrak{G}_\delta$	$\{\top + 1\}$
$\mathfrak{B}_\delta$	$\{u \mid u_0 = u_1\}$
Σ^1_1	$\{u \mid \exists y.u_0(y) = u_1(y)\}$
Π^1_1	$\{u \mid \forall y\, \exists z.u(y)(z) = 0\}$

Returning now to the example involving trees mentioned at the beginning of this section, we see that the set of perfect (finite) trees can be written in the form:

$$\{x \mid x = \mathbf{tree}(x), \mathbf{perf}(x) = 0\} = \{x \mid \langle x, \mathbf{perf}(x)\rangle = \langle \mathbf{tree}(x), 0\rangle\};$$

thus it is a $\mathfrak{B}_\delta$-set. (Note that $\mathfrak{B}_\delta$ are obviously closed under finite intersection by the ordered pair method just illustrated; that they are closed under finite union is a little messier to make explicit, but the essential idea is contained in the proof of Theorem 6.7.)

As another example, we might wish to allow infinite trees but not the strange tree $\top$. Consider the following function:

$$\mathbf{top} = \lambda u : \mathbf{tree}.\mathbf{which}(u) \subseteq \bot, \mathbf{top}(\mathbf{fst}(\mathbf{out}(u))) \cup \mathbf{top}(\mathbf{snd}(\mathbf{out}(u))). \tag{6.4}$$

We can show that $\mathbf{top} : \mathbf{P}\omega \to \{\bot, \top\}$. For a tree u the equation $\mathbf{top}(u) = \bot$ means that it *does not* contain $\top$, or as we might say: it is *topless*. The topless trees form a closed subset of the subspace of trees. (An interesting retract is the function $\lambda u.\mathbf{tree}(u) \cup \mathbf{top}(u)$ whose range consists exactly of the topless trees plus *one* exceptional tree $\top$.) Such a closed subset of (the range of) a retract is a kind of *semilattice*. (We shall not introduce a precise definition here.) Every directed subset has a limit (least upper bound) and every pair with an upper bound has a least upper bound. But generally least upper bounds do not have to exist within the semilattice. The type of domains that interest us *become* continuous lattices with the *addition* of a top element $\top$ larger than all the other elements. The elimination of $\top$ is done with a function like **top** of our example. This is convincing evidence to the author that an independent theory of semilattices is quite unnecessary: they can all be *derived* from lattices. The problem is simply to define the top-cutting operation, then restriction to the "topless" elements is indicated by an equation (like $\mathbf{top}(u) = \bot$). In this way all the constructions are kept within the control of a smooth-running theory based on LAMBDA. This point seems to be important if one wants to keep track of which functions are computable.

An aspect of the problem of classification treated in this section which has not been given close enough attention is the explicitly constructive way of verifying the closure properties of the classes. Consider the class $\mathfrak{B}_\delta$, for example. Let B be the typical set as shown in Table 3. Then whatever $f \in \mathbf{P}\omega$ we choose, the set

$$\{x \mid f(x) \in B\}$$

is a $\mathfrak{B}_\delta$-set and every such set has this form. Thus the f's *index* the elements of the class. Suppose $f, g \in \mathbf{P}\omega$. What we should look for are two LAMBDA-definable combinators such that **union**$(f)(g)$ and **inter**$(f)(g)$ give the functions that index the union and intersection of the sets determined by f and g. That is, we want:

$$\{x|\mathbf{union}(f)(g)(x) \in B\} = \{x|f(x) \in B\} \cup \{x|g(x) \in B\}.$$

It should be possible to extract the precise definition from the outline of the proofs given above, but in general this matter needs more investigation. There may very well be certain classes where such operations are not constructive, even though the classes are simply defined.

7. Total functions and functionality. There is an inevitable conflict between the concepts of *total* and *partial* functions: we desire the former, but it is the latter we usually get. Total functions are better because they are "well-defined" at all their arguments, but the rub is that there is no general way of deciding when a *definition* is going to be well-defined in all its uses. In analysis we have singularities, and in recursion theory we have endless, nonfinishing computations. In the present theory we have in effect evaded the question in two ways. First we have embraced the partial function as the norm. But secondly, and possibly confusingly, the multiple-valued functions are normal, total functions from $\mathbf{P}\omega$ into $\mathbf{P}\omega$. The point, of course, is that we are making a *model* of the partial functions in terms of ordinary mathematical functions. But note that the success of the model lies in *not* using arbitrary functions: it is only the continuous functions that correspond to the kind of partial functions we wanted to study. It would be a mistake to think of the variables in λ-calculus as ranging over arbitrary functions—and this mistake was made by both Church and Curry. The fixed-point operator **Y** shows that we must restrict attention to functions which do have fixed points. It is certainly the case that $\mathbf{P}\omega$ is not the only model for the λ-calculus, but it is a very satisfactory model and is rich enough to illustrate what can and what cannot be done with partial functions.

Whatever the pleasures of partial functions (and the multiple-valued ones, too), the desire for total functions remains. Take the integers. We are more interested in ω than $\omega \cup \{\bot, \top\}$. Since the multiple values $\bot$ and $\top$ are but two in number, it is easy to avoid them. The problem becomes tiresome in considering functions, however. The lattice represented by the retract $\mathbf{int} \multimap \mathbf{int}$ is much too large, in that there are as many nontotal functions in this domain as total ones. The aim of the present section is to introduce an interpretation of a theory of functionality in the model $\mathbf{P}\omega$ that provides a convenient way of restricting attention to the functions (or other objects) that are total in the desired sense. The theory of functionality is rather like proposals of Curry, but not quite the same for important reasons as we will see.

In the theory of retracts of §§ 5 and 6, the plan of "restricting attention" was the very simple one of restricting to a *subset*. It was made notationally simple as the subsets in question could be parameterized by continuous functions. The retraction mappings stand in for their ranges. Even better, certain continuous functions act on these retractions as space-forming functors (such as $\oplus$ and $\multimap$), which gives greater notational simplicity because one language is able to serve for several

tasks. When we pass to the theory of total functions, this same kind of simplicity is no longer possible owing to an increase in quantifier complexity in the necessary definitions. (This remark is made definite below.) Another point where there is some loss of simplicity concerns the representation of entities in $\mathbf{P}\omega$: subsets will no longer be enough, since we will need *quotients* of subsets. This is not a very startling point. Many constructions are affected in a natural way via equivalence classes. An equivalence relation makes you blind to certain distinctions. It may be easier also to remain a bit blind than to search for the most beautiful representative of an equivalence class: there may be nothing to choose between several candidates, and it can cost too much effort to attempt a choice. Thus our first agreement is that for many purposes a *kind* of object can be taken as a set of equivalence classes for an equivalence relation on a subset of $\mathbf{P}\omega$.

Because $\mathbf{P}\omega$ is closed under the pairing function $\lambda x \lambda y.\langle x, y\rangle$, we shall construe relations on subsets of $\mathbf{P}\omega$ *as* subsets of $\mathbf{P}\omega$ all of whose elements are ordered pairs. That is, a relation A satisfies this inclusion:

$$A \subseteq \{\langle x, y\rangle | x, y \in \mathbf{P}\omega\}. \tag{7.1}$$

DEFINITION. A (restricted) *equivalence relation* on $\mathbf{P}\omega$ is a symmetric and transitive relation on $\mathbf{P}\omega$.

Such relations are restricted because they are only reflexive on their domains—which are the same as their ranges—and these are the subsets with which the relations are concerned. We shall write $x\,A\,y$ for $\langle x, y\rangle \in A$ and $x : A$ for $x\,A\,x$. What we assume about these relations is the following:

$$x\,A\,y \quad \text{implies} \quad y\,A\,x. \tag{7.2}$$

$$x\,A\,y \quad \text{and} \quad y\,A\,z \quad \text{imply} \quad x\,A\,z. \tag{7.3}$$

In case a is a retract, we introduce an equivalence relation to correspond:

$$E_a = \{\langle x, x\rangle | x : a\}. \tag{7.4}$$

This is the identity relation restricted to the range of a. Such relations (for obvious reasons) and many others satisfy an additional *intersection property*:

$$x\,A\,y \quad \text{and} \quad x\,A\,z \quad \text{imply} \quad x\,A\,(y \cap z). \tag{7.5}$$

We shall not generally assume (7.5) in this short discussion, but it is often convenient.

Each equivalence relation represents a *space*: the space of all its equivalence classes. Such spaces form a category more extensive than the category of retracts studied above. The familiar functors can be extended to this larger category by these definitions:

$$A \to B = \{\langle \lambda x.u(x), \lambda x.v(x)\rangle | u(x) B v(y) \text{ whenever } x\,A\,y\}, \tag{7.6}$$

$$A \times B = \{\langle\langle x, x'\rangle, \langle y, y'\rangle\rangle | x\,A\,y \text{ and } x'\,B\,y'\}, \tag{7.7}$$

$$A + B = \{\langle\langle 0, x\rangle, \langle 0, y\rangle\rangle | x\,A\,y\} \cup \{\langle\langle 1, x'\rangle, \langle 1, y'\rangle\rangle | x'\,B\,y'\}. \tag{7.8}$$

THEOREM 7.1 (The closure theorem). *If A and B are restricted equivalence relations, then so are $A \to B$, $A \times B$ and $A + B$. We find*:

(i) $f : A \to B$ *iff* $f = \lambda x.f(x)$ *and whenever* $x\, A\, y$, *then* $f(x)\, B f(y)$, *in particular*:

(ii) *if* $f : A \to B$ *and* $x : A$, *then* $f(x) : B$; *furthermore*,

(iii) $u : A \times B$ *iff* $u = \langle u_0, u_1 \rangle$ *and* $u_0 : A$ *and* $u_1 : B$;

(iv) $u : A + B$ *iff either* $u = \langle 0, u_1 \rangle$ *and* $u_1 : A$ *or* $u = \langle 1, u_1 \rangle$ *and* $u_1 : B$.

It follows easily from 7.1 that the restricted equivalence relations form a Cartesian closed category which—in distinction to the category of retracts—has disjoint sums (or *coproducts* as they are usually called in category theory). This result is probably a special case of a more general theorem. The point is that $\mathbf{P}\omega$ itself is a space in a Cartesian closed category (that of continuous lattices and continuous maps) and it contains as subspaces the Boolean space and especially its own function space and Cartesian square. In this circumstance any such rich space must be such that its restricted equivalence relations again form a good category. Our construction is not strictly categorical in nature, as we have used the *elements* of $\mathbf{P}\omega$ and have relied on being able to form arbitrary subsets (arbitrary relations). But a more abstract formulation must be possible. The connection with the category of retracts is indicated in the next theorem.

THEOREM 7.2 (The isomorphism theorem). *If a and b are retracts, we have the following isomorphisms and identities relating the spaces*:

(i) $E_a \cong \{\langle x, y \rangle | a(x) = a(y)\}$;

(ii) $E_{a \to b} \cong E_a \to E_b$;

(iii) $E_{a \otimes b} = E_a \times E_b$;

(iv) $E_{a \oplus b} = E_a + E_b \cup \{\langle \bot, \bot \rangle, \langle \top, \top \rangle\}$.

Part (iv) is not categorical in nature as it stands, but (ii) amd (iii) indicate that E is a functor from the category of retracts into the category of equivalence relations that shows that the former is a full sub-Cartesian-closed category of the latter. We cannot pursue the categorical questions here, but note that there are many subcategories that might be of interest; for example, the equivalence relations with the intersection property are closed under $\to$, $\times$, and $+$.

Returning to the question of total functions we introduce this notation:

$$N = \{\langle n, n \rangle | n \in \omega\}. \tag{7.9}$$

This is the type of the integers *without* and $\bot$ and $\top$, i.e., the total integers. We note that:

$$N = \{u | u = \langle u_0, u_0 \rangle,\ u_0 = \mathbf{int}(u_0),\ \mathbf{mid}(u_0) = 0\}. \tag{7.10}$$

Thus N is a $\mathfrak{B}_\delta$-set. What is $N \to N$? We see:

$$f : N \to N \quad \text{iff} \quad f : \mathbf{fun} \text{ and } f(n) \in \omega \text{ whenever } n \in \omega. \tag{7.11}$$

This $N \to N$ is indeed the type of *all* total functions from ω into ω. It can be shown that $N \to N$ is also a $\mathfrak{B}_\delta$-set: good. But what is $(N \to N) \to N$? This is no longer a $\mathfrak{B}_\delta$-set, the best we can say is Π^1_1. By Theorem 7.1 it corresponds to the type of all

(extensional) *continuous* total functions from $N \to N$ into N. (The condition on A and B on the right side of (7.6) makes the concept of function embodied in $A \to B$ extensional, since the functions are meant to preserve the equivalence relations.)

A more precise discussion identifies $N \to N$ as a topological space, usually called the *Baire space.* If we introduce the finite discrete spaces by:

$$N_k = \{\langle n, n\rangle | n < k\}, \tag{7.12}$$

then $N \to N_2$ can also be identified with a topological space, usually called the Cantor space. In this identification we find at the next type, say either $(N \to N) \to N$ or $(N \to N_2) \to N_2$, that elements correspond to the usual notion of continuous function defined in topological terms. However, these higher type spaces are not at all conveniently taken as *topological* spaces. Certain of them can be identified as *limit spaces* according to the work of Hyland, and for these $\to$, $\times$, and $+$ have the natural interpretation. We cannot enter into these details here, but we can remark that the higher type spaces become ever more complicated. Thus $((N \to N) \to N) \to N$ is a Π^1_2-set and each $\to$ will add another quantifier to the definition. This is reasonable, because to say that a function is total is to say that *all* its values are well-behaved. But if its domain is a complex space, this statement of totality is even more complex. Despite this complexity, however, it is possible to sort out what kind of mapping properties many functions have. We shall mention a few of the combinators.

THEOREM 7.3 (The functionality theorem). *The combinators* **I**, **K**, *and* **S** *enjoy the following functionality properties which hold for all equivalence relations* A, B, C:

(i) $\mathbf{I} : A \to A$;

(ii) $\mathbf{K} : A \to (B \to A)$;

(iii) $\mathbf{S} : (A \to (B \to C)) \to ((A \to B) \to (A \to C))$.

Furthermore, these combinators are uniquely determined by these properties.

Let us check that S satisfies (iii). Suppose that:

$$f(A \to (B \to C))f'.$$

We must show that:

$$\mathbf{S}(f)((A \to B) \to (A \to C))\mathbf{S}(f').$$

To this end suppose that:

$$g(A \to B)g'.$$

We must show that:

$$\mathbf{S}(f)(g)\,(A \to C)\,\mathbf{S}(f')(g').$$

To this end suppose that:

$$x\,A\,x'.$$

We must show that:

$$\mathbf{S}(f)(g)(x)C\mathbf{S}(f')(g')(x').$$

Now by definition of the combinator **S** we have:

$$\mathbf{S}(f)(g)(x) = f(x)(g(x)),$$
$$\mathbf{S}(f')(g')(x') = f'(x')(g'(x')).$$

By assumptions on g, g' and on x, x', we know:

$$g(x)\ B\ g'(x').$$

By assumptions on f, f' and on x, x', we know:

$$f(x)(B \to C)f'(x').$$

The desired conclusion now follows when we note such combinations as $\mathbf{S}(f)$ and $\mathbf{S}(f)(g)$ are indeed functions. (We are using Theorem 7.1(i) several times in this case.)

In the case of the converse, let us suppose by way of example that $k \in \mathbf{P}\omega$ is such that

$$k : A \to (B \to A)$$

holds for all equivalence relations A and B. By specializing to, say, the identity relation we see that whatever $a \in \mathbf{P}\omega$ we take, both k and $k(a)$ are functions. To establish that $k = \mathbf{K}$ we need to show that the equation:

$$k(a)(b) = a$$

holds for all $a, b \in \mathbf{P}\omega$. This is easy to prove, for we have only to set:

$$A = \{\langle a, a\rangle\} \quad \text{and} \quad B = \{\langle b, b\rangle\},$$

and the equation follows at once. Not all proofs are quite so easy, however.

In the case of the combinator **S** it is not strictly true to say that Theorem 7.3(iii) determines it outright. The exact formulation is this: *if* $s \in \mathbf{P}\omega$ is such that:

$$s(f) = s(\lambda x \lambda y.f(x)(y)) \quad \text{and} \quad s(f)(g) = s(f)(\lambda x.g(x))$$

for all $f, g \in \mathbf{P}\omega$; and *if*

$$s : (A \to (B \to C)) \to ((A \to B) \to (A \to C))$$

for all A, B, C, *then* $s = \mathbf{S}$. In other words, we need to know that s converts its first two arguments into functions with the right number of places before we can say that its explicit functionality identifies as being the combinator **S**.

In Hindley, Lercher and Seldin (1972) they show that the functionality property:

$$\lambda f \lambda g.f \circ g : (B \to C) \to ((A \to B) \to (A \to C)), \tag{7.13}$$

follows from Theorem 7.1(ii) and Theorem 7.3(ii) and (iii) in view of the identity:

$$\lambda f \lambda g.f \circ g = \mathbf{S}(\mathbf{K}(\mathbf{S}))(\mathbf{K}) \tag{7.14}$$

(see Appendix A).

A more interesting result concerns the *iterators* defined as follows:

$$\mathbf{Z}_0 = \lambda f \lambda x.x, \tag{7.15}$$

$$\mathbf{Z}_{n+1} = \lambda f \lambda x.f(\mathbf{Z}_n(f)(x)). \tag{7.16}$$

In other words, $\mathbf{Z}_n(f)(x) = f^n(x)$. These natural combinators can be typed very easily, but Gordon Plotkin has shown that the obvious typing actually characterizes them.

THEOREM 7.4 (The iterator theorem). *The combinators* $\mathbf{Z}_n$ *enjoy the following functionality property which holds for all equivalence relations* A:

(i) $$\mathbf{Z}_n : (A \to A) \to (A \to A).$$

Further, if any element $z \in \mathbf{P}\omega$ *satisfies* (i) *for all* A, *then it must be one of the iterators, provided that* $z(f) = z(\lambda x.f(x))$ *holds for all* $f \in \mathbf{P}\omega$.

That each of the $\mathbf{Z}_n$ satisfies Theorem 7.4(i) is obvious. Suppose z were another such element. Then clearly:

$$z = \lambda f \lambda x.z(f)(x).$$

Suppose f and x are fixed for the moment. Let:

$$A = \{\langle f^n(x), f^n(x)\rangle | n \in \omega\},$$

where we can suppose in addition that:

$$f = \lambda x.f(x).$$

Then $f : A \to A$ is clear, and so $z(f) : A \to A$ also. But $x : A$, therefore $z(f)(x) = f^n(x)$, for some $n \in \omega$, because $z(f)(x) : A$. The trouble with this easy part of the argument is that the integer n *depends* on f and x. What we must show is that it is independent of f and x, then $z = \mathbf{Z}_n$ will follow.

Plotkin's method for this case is to introduce some independent successor functions:

$$\sigma_j = \lambda x.\{(j, k+1) | (j, k) \in x\}. \tag{7.17}$$

Note that:

$$\sigma_j^m((j', 0)) = \begin{cases} (j, m) & \text{if } j = j'; \\ \perp & \text{if } j \neq j'. \end{cases} \tag{7.18}$$

It then follows that:

$$(\sigma_j \cup \sigma_{j'})^m((j, 0) \cup (j', 0)) = (j, m) \cup (j', m). \tag{7.19}$$

Having these identities, we return to the argument.

From what we saw before, given $j \in \omega$, there is an n_j such that:

$$z(\sigma_j)((j, 0)) = \sigma_j^{n_j}(j, 0) = (j, n_j).$$

Take any two $j, j' \in \omega$. We also know there is an $n \in \omega$ where:

$$z(\sigma_j \cup \sigma_{j'})((j, 0) \cup (j', 0)) = (j, n) \cup (j', n),$$

in view of (7.19). But since $\sigma_j \subseteq \sigma_j \cup \sigma_{j'}$ and $\sigma_{j'} \subseteq \sigma_j \cup \sigma_{j'}$, we have:

$$(j, n_j) \cup (j', n_{j'}) \subseteq (j, n) \cup (j', n).$$

It follows that

$$n_j = n = n_{j'}$$

and so they are all equal. This determines the fixed $n \in \omega$ we want.

Suppose that both f and x are finite sets in $\mathbf{P}\omega$. Choose $j > \max(f \cup x)$. Let A this time be the least equivalence relation such that:

$$f^m(x)\, A\, f^m(x) \cup (j, m)$$

holds for all $m \in \omega$. We then check that:

$$\lambda x.f(x)(A \to A)(\lambda x.f(x)) \cup \sigma_j.$$

Therefore, we have:

$$z(\lambda x.f(x))(A \to A) z((\lambda x.f(x)) \cup \sigma_j),$$

and since $x\, A\, x \cup (j, 0)$, we get:

$$z(\lambda x.f(x))(x)\, A\, z((\lambda x.f(x)) \cup \sigma_j)(x \cup (j, 0)).$$

Now there is an integer $m \in \omega$ such that:

$$\begin{aligned} z((\lambda x.f(x)) \cup \sigma_j)(x \cup (j, 0)) &= f^m(x) \cup \sigma_j^m((j, 0)) \\ &= f^m(x) \cup (j, m), \end{aligned}$$

where we have been able to separate f and σ because j is so large. But the right-hand side must contain $z(\sigma_j)(j, 0) = (j, n)$. Thus m and our fixed n are the same. The other element

$$z(\lambda x.f(x))(x) = f^q(x)$$

for some $q \in \omega$. Thus we have:

$$f^q(x)\, A\, f^n(x) \cup (j, n).$$

Again since j is so large, $(j, n) \notin f^q(x)$. Thus by our choice of A we must have $f^q(x) = f^n(x)$. This means then, since n is fixed, that for *all finite* f, x:

$$z(\lambda x.f(x))(x) = f^n(x).$$

But then by continuity this equation holds for *all* f, x. It follows now that $z = \mathbf{Z}_n$, by the proviso of the theorem.

These results bring up many questions which we leave unanswered here. For example, which combinators (i.e., pure λ-terms) have functionality as in the examples above, and can we decide when a term is one such? In particular, can the diagonal combinator $\lambda x.x(x)$ be typed? (The argument of Hindley, et al. (1972, p. 81) is purely formal and does not apparently apply to the model.) What about terms in LAMBDA beyond the pure λ-calculus?

Appendix A. Proofs and technical remarks.

For Section 1. If we give the two-point space $\{\bot, \top\}$ the weak T_0-topology with just three open sets: $\varnothing$, $\{\top\}$, $\{\bot, \top\}$, we have what is called the Sierpinski space and its infinite product $\{\bot, \top\}^\omega$ with the product topology is the same as $\mathbf{P}\omega$. The finite sets $e_n \in \mathbf{P}\omega$ correspond exactly to the usual basic open sets for the product. For those familiar with such notions, this well-known observation makes many of the facts mentioned in this section fairly obvious. From any point of view, Theorem 1.1 and the remarks in the following paragraph are simple exercises.

Proof of Theorem 1.2. Equation (i) as a functional equation comes down to

$$\{m | \exists e_n \subseteq x. m \in f(e_n)\} = f(x),$$

which is just another way of writing the definition of continuity. Thus it is indeed true for all x. Next, inclusion (ii) means that if $(n, m) \in u$, then $\exists e_k \subseteq e_n.(k, m) \in u$. Clearly all we need to do is take $k = n$. If we also want the converse inclusion to hold, then what we need is condition (iii).

Proof of Theorem 1.3. Substitution is generalized composition of functions of many variables with all possible identifications and permutations of the variables; however, as we are able to define continuity by separating the variables, the argument reduces to a few special cases. The first trick is to take advantage of monotonicity. Thus, suppose $f(x, y)$ is continuous in each of its variables. What can we say of $f(x, x)$, a very special case of substitution? We calculate

$$\begin{aligned} f(x, x) &= \bigcup\{f(e_n, x) | e_n \subseteq x\} \\ &= \bigcup\{f(e_n, e_m) | e_n \subseteq x, e_m \subseteq x\}. \end{aligned}$$

Then if we think of $e_k = e_n \cup e_m$ and realize that $f(e_n, e_m) \subseteq f(e_k, e_k)$, we see that

$$f(x, x) = \bigcup\{f(e_k, e_k) | e_k \subseteq x\}.$$

This means that $f(x, x)$ is continuous in x. This same argument works if other variables are present, as in the passage from $f(x, y, z, w)$ to $f(x, x, z, w)$. When an identification of more than two variables is required, as from $f(x, y, z, w)$ to $f(x, x, x, x)$, the principle is just applied several times.

Finally to show that $f(g(x, y), h(y, x, y))$ is continuous, it is sufficient to show that $f(g(x, y), h(z, u, v))$ is continuous in each of its variables *separately*. By simply overlooking the remaining variables, this comes down to showing that $f(g(x))$ is continuous if f and g are. But the proof for ordinary composition is very easy with the aid of the characterization theorem (Theorem 1.1).

Proof of Theorem 1.4. This well-known fact holds for continuous functions on many kinds of chain-complete partial orderings; but $\mathbf{P}\omega$ illustrates the idea well enough. Suppose f had a fixed point $x = f(x)$. Then since $\varnothing \subseteq x$ and f is monotonic, we see that $f(\varnothing) \subseteq f(x) = x$. But then again, $f(f(\varnothing)) \subseteq f(x) = x$; and so by induction, $f^n(\varnothing) \subseteq x$. This proves that $\mathbf{fix}(f) \subseteq x$; and thus if $\mathbf{fix}(f)$ is a fixed point, it must be the least one. To prove that it *is* a fixed point, we need a fact that will often be useful:

LEMMA. *If $x_n \subseteq x_{n+1}$ for all n, and if f is continuous, then*

$$f(\bigcup\{x_n | n \in \omega\}) = \bigcup\{f(x_n) | n \in \omega\}.$$

Proof. By monotonicity, the inclusion holds in one direction. Suppose $e_m \subseteq f(\bigcup\{x_n | n \in \omega\})$. Then by Theorem 1.1 we have $e_m \subseteq f(e_k)$ for some $e_k \subseteq \bigcup\{x_n | n \in \omega\}$. Because e_k is finite and the sequence is increasing, we can argue that $e_k \subseteq x_n$ for some n. But then $f(e_k) \subseteq f(x_n)$. This shows that $e_m \subseteq \bigcup\{f(x_n) | n \in \omega\}$ and proves the inclusion in the other direction. (*Exercise*: Does this property characterize continuous functions?)

Proof of Theorem 1.4 *concluded.* Noting that $f^n(\varnothing) \subseteq f^{n+1}(\varnothing)$ holds for all n, we can calculate:

$$f(\mathbf{fix}(f)) = \bigcup\{f(f^n(\varnothing)) | n \in \omega\} = \bigcup\{f^{n+1}(\varnothing) | n \in \omega\}.$$

But this is just $\mathbf{fix}(n)$, since the only term left out is $f^0(\varnothing) = \varnothing$.

Proof of Theorem 1.5. The function $\bar{f}$ is clearly well-defined even when $y \in Y$ has a neighborhood U where $X \cap U = \varnothing$: in that case $\bar{f}(y) = \omega$ by convention on the meaning of $\bigcap$ in $\mathbf{P}\omega$. In case $x \in X$, it is obvious that $\bar{f}(x) \subseteq f(x)$. For the opposite inclusion, suppose that $m \in f(x)$. Because f is continuous and $\{z | m \in z\}$ is open in $\mathbf{P}\omega$, there is an open subset V of X such that $x' \in V$ always implies that $m \in f(x')$. But X is a subspace of Y, so $V = X \cap U$ for some open subset U of Y. Thus we can see why $m \in \bar{f}(x)$. It remains to show that $\bar{f}$ is itself continuous.

We must show that the inverse image under $\bar{f}$ of every open subset of $\mathbf{P}\omega$ is open in Y. But the open subsets of $\mathbf{P}\omega$ are unions of finite intersections of sets of the form $\{z | m \in z\}$. Thus it is enough to show that $\{y | m \in \bar{f}(y)\}$ is always open in Y. But this set equals $\bigcup\{U | m \in \bigcap\{f(x) | x \in X \cap U\}\}$, which being a union of open sets is open. Note that what we have proved is that $\bar{f}$ is continuous no matter what function f is given; however, if f is not continuous, then $\bar{f}$ cannot be an extension of f.

For readers not as familiar with general topology we note that the idea of Theorem 1.5 can be turned into a *definition.* Suppose $X \subseteq \mathbf{P}\omega$ is a sub*set* of $\mathbf{P}\omega$. It becomes a sub*space* with the relative topology. What are the continuous functions $f: X \to \mathbf{P}\omega$? From Theorem 1.5 we see that a necessary and sufficient condition is that the $\bar{f}: \mathbf{P}\omega \to \mathbf{P}\omega$ *be* an extension of f. Thus for $x \in X$ we can write the equation $f(x) = \bar{f}(x)$ as a biconditional:

$$m \in f(x) \quad \text{iff} \quad \exists e_n \subseteq x \; \forall x' \in X[e_n \subseteq x' \text{ implies } m \in f(x')],$$

which is to hold for all $m \in \omega$. This form of the definition of continuity on a subspace is more complicated than the original definition, because in general $e_n \notin X$ and we cannot write $f(e_n)$.

Proof of Theorem 1.6. What T_0 means is that every point of X is uniquely determined by its neighborhoods. Now $\varepsilon(x)$ just tells you the set of indices of the (basic) neighborhoods of x. Thus it is clear that ε is one-to-one. To prove that it is continuous, we need only note:

$$\{x | n \in \varepsilon(x)\} = U_n,$$

which is always open. To show that ε is an embedding, we must finally check that the *images* of the open sets U_n are open in $\varepsilon(X)$. This comes down to showing:

$$\varepsilon(U_n) = \varepsilon(X) \cap \{z | n \in z\},$$

which is clear.

For Section 2. Equation (2.1) defines a continuous function because it is a special case of Theorem 1.5, where we have been able to simplify the definition into cases because ω is a very elementary subset of $\mathbf{P}\omega$. Equation (2.2) gives a continuous function since the definition makes $\hat{p}$ distributive, as remarked in the text for finite unions, but it is just as easy to show that $\hat{p}$ distributes over arbitrary unions. The difference between a continuous f and a distributive $\hat{p}$ is this: to find $m \in f(x)$ we need a finite subset $e_n \subseteq x$ with $m \in f(e_n)$; however, to find $m \in \hat{p}(x)$ we need only *one* element $n \in x$ with $m \in \hat{p}(\{n\}) = \hat{p}(n) = p(n)$. Continuous functions are generalizations of distributive functions. The generality is necessary. For example, in (2.3) we see another function x; y distributive in each of its variables; but take care: the function x; x is not distributive in x—there is no closure under substitution. This is just one reason why continuous functions are better. Another good example comes from (2.4) if you compare the functions x, $x+x$, $x+x+x$, etc.

Equations (2.5)–(2.7) are very elementary. Note that $z \supset x$, y is distributive in each of its variables. We could write: $z \supset x$, $y = \hat{p}(z)$, where $p(0) = x$ and $p(n+1) = y$, to show that it is distributive in z.

Proof of Theorem 2.1. If we did not use the λ-notation, then all LAMBDA-definable functions would be obtained by substitution from the first five (cf. Table 2). Since they are all seen to be continuous, the result would then follow by the substitution theorem (Theorem 1.3). Bringing in λ-abstraction means that we have to combine Theorem 1.3 with this fact:

LEMMA. *If $f(x, y, z, \cdots)$ is a continuous function of all its variables, then $\lambda x.f(x, y, z, \cdots)$ is a continuous function of the remaining variables.*

Proof. It is enough to consider one extra variable. We compute from the definition of λ in Table 2 as follows:

$$\begin{aligned}\lambda x.f(x, y) &= \{(n, m) | m \in f(e_n, y)\} \\ &= \{(n, m) | \exists e_k \subseteq y.m \in f(e_n, e_k)\} \\ &= \bigcup\{\{(n, m) | m \in f(e_n, e_k)\} | e_k \subseteq y\} \\ &= \bigcup\{\lambda x.f(x, e_k) | e_k \subseteq y\}.\end{aligned}$$

Thus $\lambda x.f(x, y)$ is continuous in y.

Proof of Theorem 2.2. The reason behind this result is the restriction to continuous functions. Theorem 2.1 shows that we cannot violate the restriction by giving definitions in LAMBDA, and the graph theorem (Theorem 1.2) shows that continuous functions correspond perfectly with their graphs.

The verification of (α) of Table 1 is obvious as the 'x' in '$\lambda x.\tau$' is a bound variable. (Care should be taken in making the proviso that 'y' is not otherwise free in τ.) The same would of course hold for any other pair of variables. We do not bother very much about alphabetic questions.

The verification of (β) is just a restatement of Theorem 1.2(i). Let τ define a function f (of x). Then by definition $f(x) = \tau$ and $\lambda x.\tau = \mathbf{graph}(f)$. Also $\mathbf{fun}(u)(x)$ in the notation of Theorem 1.2 is the same as the binary operation $u(x)$ in the notation of LAMBDA. Thus in Theorem 1.2(i) if we apply both sides to y we get nothing else than (β).

Half of property (ξ) is already implied by (β): the implication from left to right. (Just apply both sides to x.) In the other direction, $\forall x.\tau = \sigma$ means that τ and σ define the *same* function of x; thus, the two graphs must be equal.

Remarks on other laws. The failure of (η) simply means that not every set in $\mathbf{P}\omega$ *is* the graph of a function. Condition (iii) of Theorem 1.2 is equivalent to saying that $u = \lambda x.u(x)$, in other words, u is the graph of some function if and only if it is the graph of the function determined by u.

Law (μ) is the monotone property of application (in both variables); therefore, (μ) and (ξ) together imply (ξ^*) from left to right. Suppose that $\forall x.\tau \subseteq \sigma$; then clearly:

$$\{(n, m) | m \in \tau[e_n/x]\} \subseteq \{(n, m) | m \in \sigma[e_n/x]\},$$

which gives (ξ^*) from right to left.

There are, by the way, other laws valid in the model, as explained in the later results.

Proof of Theorem 2.3. This is a standard result combinatory logic. We have only to put:

$$u = \lambda x_0 \lambda x_1 \cdots \lambda x_{n-1}.f(x_0, x_1, \cdots, x_{n-1}).$$

That is, we use the iteration of the process of forming the graph of a continuous function. As each step (from the inside out) keeps everything continuous, we are sure that the equation of Theorem 2.3 will hold for iterated application.

Proof of Theorem 2.4. This can be found in almost any reference on combinatory logic or λ-conversion. The main idea is to eliminate the λ in favor of the combinators. The fact that we have a few other kinds of terms causes no problem if we introduce the corresponding combinators. The method of proof is to show, for any LAMBDA-term τ with free variables among $x_0, x_1, \cdots, x_{n-1}$, that there is a combination γ of combinators such that:

$$\tau = \gamma(x_0)(x_1) \cdots (x_{k-1}).$$

This can be done by induction on the complexity of τ.

Proof of Theorem 2.5. The well-known calculation shows that we have from (2.8):

$$\mathbf{Y}(u) = (\lambda x.u(x(x)))(\lambda x.u(x(x))) = u(\mathbf{Y}(u)).$$

Thus $\mathbf{Y}(u)$ *is* a fixed point of the function $u(x)$. What is needed is the proof to show that it is the least one.

Let $d = \lambda x.u(x(x))$ and let a be any other fixed point of $u(x)$. To show, as we must, that $d(d) \subseteq a$, it is enough to show that $e_l \subseteq d$ always implies $e_l(e_l) \subseteq a$; because by continuity we have:

$$d(d) = \bigcup \{e_l(e_l) | e_l \subseteq d\}.$$

By way of induction, suppose that this implication holds for all $n < l$. Assume that $e_l \subseteq d$ and that $m \in e_l(e_l)$. We will want to use the induction hypothesis to show that $m \in a$. By the definition of application, there exists an integer n such that $(n, m) \in e_l$ and $e_n \subseteq e_l$. But $n \leqq (n, m) < l$, and $e_n \subseteq d$. By the hypothesis, we have $e_n(e_n) \subseteq a$. Note that $(n, m) \in d$ also, and that d is defined by λ-abstraction; thus,

$m \in d(e_n)$ by definition. By monotonicity $u(e_n(e_n)) \subseteq u(a) = a$; therefore $m \in a$. This shows that $e_l(e_l) \subseteq a$, and the inductive proof is complete.

Remark. Note that we did not actually use the fixed-point theorem in the proof, but we did use rather special properties of the pairing (n, m) and the finite sets e_l.

Equation (2.9) is proved easily from the definition of application; indeed $u(x)$ is distributive in u. Equation (2.10) is proved even more easily from the definition of λ-abstraction. For (2.11), we see that the inclusion from left to right would hold in general by monotonicity. In the other direction, suppose $m \in f(x) \cap g(x)$. Then for suitable k and l we have $(k, m) \in f$ and $e_k \subseteq x$, also $(l, m) \in g$ and $e_l \subseteq x$. Let $e_n = e_k \cup e_l \subseteq x$. Because f and g are *graphs*, we can say $(n, m) \in f \cap g$; and thus $m \in (f \cap g)(x)$. This is the only point where we require the assumption on graphs. Equation (2.12) follows directly from the definition of abstraction. For (2.13), which generalizes (2.9), we can also argue directly from the definition of application. In the case of intersection it is easy to find u_n such that $0 \in u_n(\top)$ for all n, but $\bigcap\{u_n | n \in \omega\} = \bot$.

Equation (2.14) is obvious because the least fixed point of the identity function must be $\bot$. A less mysterious definition would be $\bot = 0 - 1$, but the chosen one is more "logical".

For (2.15) we note that by definition:

$$\lambda z.0 = \{(n, m) | m \in 0\} = \{(n, 0) | n \in \omega\}.$$

Because $0 = (0, 0)$ and $1 = (1, 0)$, we get the hint. Equation (2.16) makes use of $\cup$ for iteration. If $x = 0 \cup (x + 1)$, then x must contain all integers; hence $x = \top$. The iteration for $\cap$ in (2.17) is more complex. The fundamental equation we need is:

$$x \cap y = x \supset (y \supset 0, \bot), ((x - 1) \cap (y - 1)) + 1.$$

This says to compute the intersection of two sets x and y, we first test whether $0 \in x$. If so, then test whether $0 \in y$. If so, then we know $0 \in x \cap y$. In the meantime we begin testing x for positive elements. If we could compute (by the same program) the intersection $(x - 1) \cap (y - 1)$, then we would get the positive elements of the intersection $x \cap y$ by adding one. This is a very slow program, but we can argue by induction that it gives us all the desired elements. Of course, $\cap$ is the least function satisfying this equation.

In the case of (2.18) it is clear that we have:

$$\lambda x.\bot = \{(n, m) | m \in \bot\} = \bot;$$

$$\lambda x.\top = \{(n, m) | m \in \top\} = \top;$$

because in the last every integer is a (number of a) pair. Suppose now that $a = \lambda x.a$ and $a \neq \top$. Let k be the least integer where $k \notin a$. Now $k = (n, m)$ for some n and m. If $m \in a$, then $(n, m) \in a = \lambda x.a$; hence $m \notin a$. But $m \leqq k$ and k is minimal; therefore, $m = k$. But this is only possible if $k = m = n = 0$. Suppose further $a \neq \bot$ and that l is the least integer where $l \in a$. Now $l = (i, j)$ with $j \in a$ and $j \leqq l$. So $j = l$ and $l = j = i = 0$. This contradiction proves that $a = \bot$ or $a = \top$.

Equations (2.19)–(2.22) are definitions, and (2.23) is proved easily by induction on i. Equation (2.24) is also a definition. To prove (2.25) we note that

$\{u_i | i \in x\}$ is continuous (even: distributive) in u and x. Thus, there is a continuous function $\mathbf{seq}(u)(x)$ giving this value. What is required is to prove that it is LAMBDA-definable. We see:

$$\mathbf{seq}(u)(x) = \{n \in u_0 | 0 \in x\} \cup \{m \in \bigcup \{u_{i+1} | i+1 \in x\} | \exists k.k+1 \in x\}$$
$$= x \supset u_0, \mathbf{seq}(\lambda t.u_{t+1})(x-1);$$

that is, **seq** satisfies the fixed-point equation for \$. Thus $\$ \subseteq \mathbf{seq}$. To establish the other inclusion we argue by induction on i for:

$$\forall x, u[i \in x \Rightarrow u_i \subseteq \$(u)(x)]$$

This is easy by cases using what we are given about \$ in (2.24); it implies that $\mathbf{seq} \subseteq \$$. Note that:

$$\lambda n \in \omega.\tau = \lambda n \in \omega.\sigma \quad \text{iff} \quad \forall n \in \omega.\tau = \sigma.$$

For primitive recursive functions, even of several variables, there is no trouble in transcribing into LAMBDA-notation any standard definition—especially as we can use the abstraction operator $\lambda n \in \omega$. If we recall that every r.e. set a has the form:

$$a = \{m | \exists n.p(n) = m+1\},$$

where p is primitive recursive, we then see that $a = \hat{p}(\top) - 1$. This means that every r.e. set is LAMBDA-definable.

Proof of Theorem 2.6. In case of a function of several variables, we remark:

$$\lambda x_0 \lambda x_1 \cdots \lambda x_{k-1}.f(x_0)(x_1) \cdots (x_{k-1})$$
$$= \{(n_0, (n_1, (\cdots, (n_{k-1}, m) \cdots))) | m \in f(e_{n_0})(e_{n_1}) \cdots (e_{n_{k-1}})\}.$$

This makes the implication from (i) to (ii) obvious. Conversely, if a *graph* u is r.e., then from the definition of application we have:

$$m \in u(e_{n_0})(e_{n_1}) \cdots (e_{n_k}) \quad \text{iff} \quad (n_0, (n_1, (\cdots, (n_{k-1}, m) \cdots))) \in u,$$

which is r.e. in $m, n_0, n_1, \cdots, n_{k-1}$. Therefore (i) and (ii) are equivalent.

We have already proved that (ii) implies (iii). For the converse we have only to show that all LAMBDA-definable sets are r.e. For this argument we could take advantage of the combinator theorem, (Theorem 2.4). Each of the six combinators are r.e., and there is no problem of showing that if u and x are r.e., then so is $u(x)$; because it is defined in such an elementary way with existential and bounded universal number quantifiers and with membership in u and in x occurring positively. Explicitly we have:

$$m \in u(x) \quad \text{iff} \quad \exists e_n \subseteq x.(n, m) \in u$$
$$\text{iff} \quad \exists n \, \forall m < n[[m \in e_n \text{ implies } m \in x] \text{ and } (n, m) \in u].$$

For Section 3. For the proof of (3.1) we distinguish cases. In case $x = y = \perp$, we note that $\mathbf{cond}(\perp)(\perp) = \perp$ and $\perp(\perp) = \perp$, so the equation checks in this case. Recall:

$$\mathbf{cond}(x)(y) = \lambda z.z \supset x, y = \{(n, m) | m \in (e_n \supset x, y)\}.$$

We can show $0 \not\in \mathbf{cond}(x)(y)$. Note first $0 = (n, m)$ iff $n = 0 = m$; furthermore, $e_0 = \perp$ and $\perp \supset x$, $y = \perp$; but $0 \not\in \perp$. Also we have:

$$\mathbf{cond}(x)(y)(0) = x \quad \text{and} \quad \mathbf{cond}(x)(y)(1) = y;$$

so if either $x \neq \perp$ or $y \neq \perp$, then $\mathbf{cond}(x)(y) \neq \perp$. In this case, $\mathbf{cond}(x)(y)$ must contain *positive* elements. The result now follows.

Theorem 3.1 is obvious from the construction of **G**, because $\mathbf{G}(\mathbf{G}) = 0$ and $\mathbf{G}(0)(0) = \mathbf{suc}$, and so the $\mathbf{G}(0)(i)$ give us all the other combinators.

The primitive recursive functions needed for (3.4)–(3.6) are standard. Equation (3.7) is a definition—if we rewrote it using the **Y**-operator—and the proof of Theorem 3.2 is easy by induction. There is also no difficulty with (3.8)–(3.12). The idea of the proof of Theorem 3.3 is contained in the statement of the theorem itself. The proof of Theorem 3.4 is already outlined in the text.

Proof of Theorem 3.5. The argument is essentially the original one of Myhill–Shepherdson. Suppose p is computable, total and extensional. Define:

$$q = \{(j, m) | m \in \mathbf{val}(p(\mathbf{fin}(j)))\},$$

where **fin** is primitive recursive, and for all $j \in \omega$:

$$\mathbf{val}(\mathbf{fin}(j)) = e_j.$$

Certainly $q \in \mathbf{RE}$, and we will establish the theorem if we can prove "continuity":

$$\mathbf{val}(p(n)) = \bigcup \{\mathbf{val}(p(\mathbf{fin}(j))) | e_j \subseteq \mathbf{val}(n)\}.$$

We proceed by contradiction. Suppose first we have a $k \in \mathbf{val}(p(n))$, where $k \not\in \mathbf{val}(p(\mathbf{fin}(j)))$ whenever $e_j \subseteq \mathbf{val}(n)$. Pick r to be a primitive recursive function whose range is *not* recursive. Define s, primitive recursive, so that for all $m \in \omega$:

$$\mathbf{val}(s(m)) = \{j \in \mathbf{val}(n) | m \not\in \{r(i) | i \leqq j\}\}.$$

The set $\mathbf{val}(n)$ must be *infinite*, because p is extensional, and if $\mathbf{val}(n) = e_j = \mathbf{val}(\mathbf{fin}(j))$, then $k \not\in \mathbf{val}(p(\mathbf{fin}(j))) = \mathbf{val}(p(n))$. Note that $\mathbf{val}(s(m))$, as a subset of the infinite set, is *finite* if m is in the range of r; otherwise it is equal to $\mathbf{val}(n)$. Again by the extensionality of p we see that $k \in \mathbf{val}(p(s(m)))$ if and only if m is *not* in the range of r. But this puts an r.e. condition on m equivalent to a non-r.e. condition, which shows there is no such k.

For the second case suppose we have a $k \not\in \mathbf{val}(p(n))$, where for a suitable $e_j \subseteq \mathbf{val}(n)$ it is the case that $k \in \mathbf{val}(p(\mathbf{fin}(j)))$. Define:

$$t = \lambda m \in \omega . e_j \cup (\mathbf{val}(m) \supset \mathbf{val}(n), \mathbf{val}(n)).$$

We have:

$$t(m) = \begin{cases} e_j & \text{if } \mathbf{val}(m) = \perp; \\ \mathbf{val}(n) & \text{if not.} \end{cases}$$

We choose u primitive recursive, where:

$$\mathbf{val}(u(m)) = t(m).$$

By the choice of k, and by the extensionality of p, and by the fact that $\mathbf{val}(n) \neq e_j$,

we have:

$$k \in \mathbf{val}(p(u(m))) \quad \text{iff} \quad t(m) = e_j$$
$$\text{iff} \quad \mathbf{val}(m) = \perp.$$

But this is impossible, since one side is r.e. in m and the other is not by Theorem 3.4. As both cases lead to contradiction, continuity is established and the proof is complete.

Proof of Theorem 3.6. Consider a degree $\mathbf{Deg}(a)$. This set is closed under application, because:

$$u(a)(v(a)) = \mathbf{S}(u)(v)(a),$$

and $\mathbf{S}(u)(v)$ is r.e. if both u and v are. Note that it also contains the element $\mathbf{G}$; hence, as a subalgebra, it is generated by a and $\mathbf{G}$.

Let A be any finitely generated subalgebra with generators $a'_0, a'_1, \cdots, a'_{n-1}$. Consider the element $a = \mathbf{cond}(\langle a'_0, a'_1, \cdots, a'_{n-1}\rangle)(\mathbf{G})$. As in the proof of Theorem 3.1, a generates A under application. It is then easy to see why $A = \mathbf{Deg}(a)$.

Proof of Theorem 3.7. We first establish (3.16) and (3.17):

$$\begin{aligned} L \circ \bar{u} \circ R &= \lambda x.L(\bar{u}(\langle 0, x\rangle)) \\ &= \lambda x.L(\langle 1, u, x\rangle) \\ &= \lambda x.u(x). \\ \bar{u} \circ \bar{v} \circ R &= \lambda x.\bar{u}(\bar{v}(\langle 0, x\rangle)) \\ &= \lambda x.\bar{u}(\langle 1, v, x\rangle) \\ &= \lambda x.\overline{u(v)}(x) \\ &= \overline{u(v)}. \end{aligned}$$

Now starting with any $u \in \mathbf{RE}$, we write $u = \tau$, where τ is formed from $\mathbf{G}$ by application alone. By (3.17), we can write $\bar{u}$ in terms of $\bar{\mathbf{G}}$ and $\mathbf{R}$ using only $\circ$. That is, $\bar{u}$ belongs to a special subsemigroup. In view of (3.16), we find that $\lambda x.u(x)$ belongs to that generated by $\mathbf{R}$, $\mathbf{L}$ and $\bar{\mathbf{G}}$. But

$$\mathbf{RE} \cap \mathbf{FUN} = \{\lambda x.u(x) | u \in \mathbf{RE}\},$$

and so the theorem is proved.

For Section 4. The notion of a *continuous lattice* is due to Scott (1970/71) and we shall not review all the facts here. One special feature of these lattices is that the lattice operations of meet and join ($\sqcap$ and $\sqcup$) are continuous (that is, commute with directed sups). As topological spaces, they can be characterized as those T_0-spaces satisfying the extension theorem (which we proved for $\mathbf{P}\omega$ in Theorem 1.5).

Proof of Theorem 4.1. Consider a continuous function a, and let $A = \{x | x = a(x)\}$. By the fixed-point theorem (Theorem 1.4) we know that A is nonempty and that it has a least element under $\subseteq$. Certainly A is partially ordered by $\subseteq$; further, A is closed under *directed* unions but not under arbitrary unions.

That is, A is not a complete sublattice of $\mathbf{P}\omega$ with regard to the lattice operations of $\mathbf{P}\omega$, but it could be a complete lattice on its own—if we can show sups exist. Thus, let $S \subseteq A$ be an arbitrary subset of A. By the fixed-point theorem, find the least solution to the equation:

$$y = \bigcup\{x | x \in S\} \cup a(y).$$

Clearly $x \subseteq y$ for all $x \in S$; and so $x = a(x) \subseteq a(y)$, for all $x \in S$. This means that $y = a(y)$, and thus $y \in A$. By construction, then, y is an upper bound to the elements of S. Suppose $z \in A$ is another upper bound for S. It will also satisfy the above equation; thus $y \subseteq z$, and so y is the least upper bound. A partially ordered set with sups also has infs, as is well known, and is a complete lattice.

Suppose that a is a retract. We can easily show that the fixed-point set A (with the relative topology from $\mathbf{P}\omega$) satisfies the extension theorem. For assume $f: X \to A$ is continuous, and $X \subseteq Y$ as a subspace. Now we can also regard $f: X \to \mathbf{P}\omega$ as continuous because A is a subspace of $\mathbf{P}\omega$. By Theorem 1.5 there is an extension to a continuous $\bar{f}: Y \to \mathbf{P}\omega$. But then $a \circ \bar{f}: Y \to A$ is the continuous function we want for A, and the proof is complete.

The space of retracts. Let us define:

$$\mathbf{RET} = \{a | a = a \circ a\},$$

the set of all retracts, which is a complete lattice in view of Theorem 4.1. It will be proved to be not a retract itself by showing it is not a continuous lattice; in fact, the meet operation on **RET** is not continuous on **RET**.

The proof was kindly communicated by Y. L. Ershov and rests on distinguishing some extreme cases of retracts. Call a retract *a nonextensive* if for all nonempty finite sets x we have $x \not\subseteq a(x)$. Call a retract *b finite* if all its values are finite (i.e., $b(\top)$ is finite). If a is nonextensive and b is finite, then Ershov notes that they are "orthogonal" in **RET** in the sense that $c = a \sqcap b = \bot$. The reason is that, since $c \subseteq b$, it is finite; but $c \subseteq a$, too, so $c(x) \subseteq a(x)$ for all x. Because c is a retract, we have $c(x) = c(c(x)) \subseteq a(c(x))$. As $c(x)$ is finite and a is nonextensive, it follows that $c(x) = \bot$ for all x.

This orthogonality is unfortunate, because consider the finite retracts $b_n = \lambda x.e_n$. We have here a directed set of retracts where $\bigcup\{b_n | n \in \omega\} = \lambda x.\top = \top$. If $\sqcap$ were continuous, it would follow that for nonextensive a:

$$a = a \sqcap \top = a \sqcap \bigcup\{b_n | n \in \omega\} = \bigcup\{a \sqcap b_n | n \in \omega\} = \bot,$$

showing that there are no nontrivial such a. But this is not so.

Let $\ll$ be a strict linear ordering of ω in the order type of the rational numbers. Define:

$$a(x) = \{m | \exists n \in x. m \ll n\}.$$

We see at once that a is continuous; and, because $\ll$ is transitive and dense, a is a retract. Since $\ll$ is irreflexive, it is the case for finite nonempty sets x that $\max_{\ll}(x) \notin a(x)$; hence, a is nonextensive. As $a(\top) = \top$, we find $a \neq \bot$. The proof is complete.

Note that there are many transitive, dense, irreflexive relations on ω, so there are many nonextensive retracts. These retracts, like a above, are *distributive.* A

nondistributive example is:

$$a' = \{m | \exists n,\ n' \in x.n \ll m \ll n'\}.$$

Many other examples are possible.

Proof of Theorem 4.2. The relation $\Subset$ is by the definition of retract reflexive on **RET**; it is also obviously antisymmetric. To prove transitivity, suppose $a \Subset b \Subset c$, then

$$a = a \circ b = a \circ b \circ c = a \circ c.$$

Similarly, $a = c \circ a$. Note, by the way, that $a \Subset b$ implies that the range of a is included in that of b; but that the relationship $a \subseteq b$ does not imply this fact. The relationship $a \Subset b$, however, is stronger than inclusion of ranges.

Proofs of Theorems 4.3–4.5. We will not give full details as all the parts of these theorems are direct calculations. Consider by way of example Theorem 4.3(i). We find:

$$\begin{aligned}(a \multimap b) \circ (a \multimap b) &= \lambda u.b \circ (b \circ u \circ a) \circ a \\ &= \lambda u.b \circ u \circ a \\ &= a \multimap b,\end{aligned}$$

provided that a and b are retracts. A very similar computation would verify part (iv), if one writes out the composition:

$$(a \multimap b') \circ (f \multimap f') \circ (b \multimap a')$$

and uses the equations:

$$f = b \circ f \circ a \quad \text{and} \quad f' = b' \circ f' \circ a'.$$

The main point of the proof of Theorem 4.6 has already been given in the text.

For Sections 5–7. Sufficient hints for proofs have been given in the text.

Appendix B. Acknowledgments and references. My greatest overall debt is to the late Christopher Strachey, who provided not only the initial stimulus and continuing encouragement, but also what may be termed the necessary irritation. Not being a trained mathematician, he often assumed that various operations made sense without looking too closely or rigorously at the details. This was particularly the case with the λ-calculus, which he used as freely as everyday algebra. Having repeatedly and outspokenly condemned the λ-calculus as a formal system without clear mathematical foundations, it was up to me to provide some alternative. The first suggestion was a typed system put forward in Scott (1969) (unpublished, but later developed as LCF by Robin Milner and his collaborators). Experience with the type structure of function spaces, which had come to my attention from work in recursion theory by Nerode, Platek and others, soon convinced me that there were many more similar structures than might at first be imagined. In particular, a vague idea about a space with a "dense" basis led quickly to the more direct construction, by inverse limits, of function spaces of "infinite" type that were very reasonable models of the classical "type-free"

λ-calculus (as well as many other calculi for other "reflexive domains). The details can be found in Scott (1971) and Scott (1973b). Algebra was justified, but the work in doing so was tiring and the exact connections with computability were not all that easy to describe.

In the meantime Plotkin (1972) contained suggestions for a "set-theoretical" construction of models, but not much notice was taken of the plan at the time it was circulated—perhaps owing to a fairly sketchy presentation of the precise semantics of the λ-calculus. The present paper evolved particularly from the project of making the connections with ordinary recursion theory easier to comprehend, since a satisfactory theory of computability and programming language semantics had to face this problem. The idea of using sets of *integers* for a model was first put forward by the author at a meeting at Oberwolfach at Easter in 1973 and in a more definitive form at the Third Scandinavian Logic Symposium shortly thereafter (see Scott (1975a) which is a preliminary and shorter version of this paper). The author gave a report on the model at the Bristol Logic Colloquium in July 1973, but did not submit a paper for the proceedings. A series of several lectures was presented at the Kiel Logic Colloquium in July 1974, covering most of the present paper which was distributed as a preprint at the meeting. The text (but unfortunately neither acknowledgments nor references) was printed in the proceedings (Springer Lecture Notes in Mathematics, vol. 499). In 1973 after experimentation with definitions somehow forced him into the definition of the model, the author realized that it was essentially the same as Plotkin's idea and, even more surprising, it was already implicit in a very precise form in much earlier work by Myhill–Shepherdson (1955) and Friedberg–Rogers (1959) (see also Rogers (1967)) on *enumeration operators*. What had happened was that Plotkin had not made enough tie-up with recursion theory, and the recursive people had not seen the tie-up with λ-calculus, even though they knew that one could do a lot with such operators. Actually, if the author had taken his own advice in 1971 (see *Continuous lattices*, Scott (1972a, end of § 2)), he would have seen that many spaces have their own continuous-function spaces as computable retracts, a fact which is just exactly the basis for the present construction; but instead he said: "it [the representation as a retract] does not seem to be of too much help in proving theorems."

Over the years in work on λ-calculus and programming language semantics, personal contact and correspondence with a large number of people has been very stimulating and helpful. I must mention particularly de Bakker, Barendregt, Bekic, Blikle, Böhm, Curry, Egli, Engeler, Ershov, Goodman, Hyland, Kreisel, Landin, Milne, Milner, Mosses, Nivat, Park, Plotkin, Reynolds, de Roever, Smyth, Stoy, Tang, Tennent, Wadsworth. (I apologize to those I have inadvertently left out of this list.) In the reference list a very imperfect attempt has been made to collect references directly relevant to the topics of this paper as well as pointers to related areas that may be of inspiration for future work. The list of papers is undoubtedly incomplete, and inaccurate as well, but the author hopes it may be of some use for those seeking orientation. It is a very vexing problem to keep such references up to date. Some remarks toward references and acknowledgments on the specific results in the various sections follow. Felipe Bracho deserves special thanks for help in the preparation of the final manuscript and with the reference list.

Section 1. The relevance of the "positive" or "weak" topology first came to the author's attention through the work of Nerode (1959). Continuous functionals were studied by Kleene and Platek and many other researchers in recursion theory. Monotonicity was particularly stressed by Platek (1964). The graphs and the definition of application are used in the same way by Plotkin (1972) and Rogers (1967, see p. 147). The fixed-point theorem is very well-known. See, e.g., Tarski (1955). The extension theorem was formulated by the author, but it is very similar to many results in point-set topology; it was used in a prominent way in Scott (1972a) to characterize continuous lattices. The embedding theorem is well-known; see, e.g., Čech (1966).

Section 2. The language LAMBDA is due to the author. Note in particular that Plotkin and Rogers do not define λ-abstraction, even though they know of the existence of many combinators and *could have* defined abstraction if anyone had ever asked them. In particular, they understood about conversion in many instances. The reduction and combinator theorems are well known from combinatory logic and can be found in any reference. The first recursion theorem is basic to all of elementary recursion theory; what is new here is the adaptation of David Park's proof (Park (1970c), unpublished) to the present model to show that Curry's "paradoxical" combinator actually does the recursion. The definition of computability and the definability theorem tie up the present theory with the older theory of enumeration operators.

Section 3. The idea of reduction to a few combinators is an old one in combinatory logic; the author only needed to find a small trick (formula (3.1)) which would take care of the arithmetical combinators. The ideas for the Gödel numbering and the proof of the second recursion theorem are standard, as is the proof of the incompleteness theorem. It only looks a little different since we combine arithmetic with the "type-free" combinators. The proof of the completeness theorem for definability (Theorem 3.5) is taken directly from Myhill–Shepherdson (1955). The author is indebted to Hyland for pointing this out. The subalgebra theorem is an easy reformulation of talk about enumeration degrees; for more information on such degrees consult Rogers (1967), Sasso (1975), and also Gutteridge (1971). The area is underdeveloped as compared to Turing degrees. Semigroups of combinators have been discussed by Church and Böhm.

Section 4. The notion of a *retract* is common in topology, but the idea of using them to define data types and of having a calculus of computable retracts is original with the author. Of course the connection between lattices and fixed points was known; more about lattices is to be found in Scott (1972a). The various operations on retracts and the idea of using fixed-point equations to define retracts recursively are due to the author. Applications to semantics were given in Scott (1971) for flow diagrams, and this has been followed up by many people, in particular Goguen, et al. (1975) and Reynolds (1974b).

Section 5. Algebraic lattices have been known for a long time (see Gratzer (1968)) and also closure operations (see, e.g., Tarski (1930)). It was Per Martin-Löf and Peter Hancock who suggested that they might form a "universe"; in particular the construction of $\mathbf{V}$ is essentially due to them. The limit theorem is due to the author.

Section 6. More information on the classification by notions in descriptive set theory of various subsets can be found in the work of Tang who also makes

connections with the work of Wadge. The various normal forms for the classes of sets (e.g., Table 3) are due to the author.

Section 7. Functionality has been studied for some time in combinatory logic (see, e.g., Hindley, et al. (1972) for an introduction). The author had the idea to see what it all means in the models; there are, of course, connections going back to Curry and Kleene, with functional interpretations of intuitionistic logic (cf. Theorem 7.3, which is well-known). The proof of Theorem 7.4 is due to Plotkin.

Appendix A. After the main body of the paper was written, Y. L. Ershov solved the author's problem about the space of retracts. Ershov's proof is presented after the discussion of the proof of Theorem 4.1 in this Appendix. Quite independently Hosono and Sato (1975) found almost exactly the same proof. Before corresponding with Ershov, the author was totally unaware of the connections with and the importance of Ershov's extensive work in "numeration" theory (see citations in the reference list).

Appendix B. All defects are due to the author.

REFERENCES

L. AIELLO, M. AIELLO AND R. W. WEYHRAUCH (1974), *The semantics of PASCAL in LCF*, Artificial Intelligence Lab. Memo. AIM-221, Stanford Univ., Stanford, Calif.

S. ALAGIC (1974), *Categorical theory of tree processing*, Category Theory Applied to Computation and Control, E. Manes, ed., Univ. of Mass., pp. 80–99.

M. A. ARBIB AND E. G. MANES (1974a), *Fuzzy morphisms in automata theory*, Category Theory Applied to Computation and Control, E. Manes, ed., Univ. of Mass., pp. 98–105.

——— (1974b), *Categorists' view of automata and systems*, Category Theory Applied to Computation and Control, E. Manes, ed., Univ. of Mass., pp. 62–79.

——— (1974c), *Basic concepts of category theory applicable to computation and control*, Category Theory Applied to Computation and Control, E. Manes, ed., Univ. of Mass., pp. 2–41.

J. W. DE BAKKER (1971a), *Recursive procedures*, Mathematical Centre Tracts, vol. 24, Amsterdam.

——— (1971b), *Recursion, induction and symbol manipulation*, Proc. MC-25 Informatica Symposium, Mathematical Centre Tracts, vol. 38, Amsterdam.

J. W. DE BAKKER AND W. P. DE ROEVER (1972), *A Calculus for Recursive Program Schemes*, Proc. IRIA Colloq. on Automata, North-Holland, Amsterdam.

J. W. DE BAKKER AND D. SCOTT (1969), *A theory of programs*, Unpublished notes, IBM Seminar, Vienna, 1969.

J. W. DE BAKKER (1969), *Semantics of Programming Languages*. Advances in Information Systems Science, vol. 2, Plenum Press, New York.

J. W. DE BAKKER AND L. G. L. MEERTENS (1973), *On the completeness of the inductive assertion method*, Mathematical Centre Rep. IW 12/72, Amsterdam.

——— (1974), *Fixed points in programming theory*, Foundations of Computer Science, J. W. de Bakker, ed., Mathematical Centre Tract, vol. 63, Amsterdam.

H. P. BARENDREGT (1971), *Some extensional term models for combinatory logics and λ-calculi*. Ph.D. thesis, Univ. of Utrecht.

H. BEKIĆ (1971), *Towards a mathematical theory of processes*, Tech. Rep. TR 25.125, IBM Lab., Vienna.

——— (1969), *Definable operations in general algebra, and the theory of automata and flowcharts*, Rep., IBM Lab., Vienna.

H. BEKIĆ ET AL. (1974), *A formal definition of a PL/1 subset, Parts I and II*, Tech. Rep. TR 25.139, IBM Lab., Vienna.

R. S. BIRD (1974), *Unsolvability of the inclusion problem for DBS schemas*, Rep. RCS22, Dept. of Computer Sci., Univ. of Reading.

G. BIRKHOFF (1967), *Lattice Theory*, 3rd ed., Colloquium Publications, vol. 25, American Mathematical Society, Providence, R.I.

A. BLIKLE (1971), *Algorithmically definable functions: A contribution towards the semantics of programming languages*, Dissertationes Math. Rozprawy Mat., 85.

——— (1972), *Equational languages*, Information Control, pp. 134–147.

——— (1973), *An algebraic approach to programs and their computations*, Proc. Symp. and Summer School on the Mathematical Foundations of Computer Sci., High Tatras, Czechoslovakia.

A. BLIKLE AND A. MAZURKIEWICZ (1972), *An algebraic approach to the theory of programs, algorithms, languages and recursiveness*, Proc. Internat. Sympos. and Summer School on the Mathematical Foundations of Computer Sci., Warsaw–Jablonna.

A. BLIKLE (1974), *Proving programs by sets of computations*, Proc. 3rd Symp. on the Mathematical Foundation of Computer Sci., Jadwisin, Poland; Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1974.

C. BÖHM (1968), *Alcune proprieta delle forme β-η-normal Nel λ-κ-calcolo.* Consiglio Nazionale del e Richerche: Publicazione 696 dell' Instituto per le Applicazioni del Calcolo, Roma.

——— (1966), *The CUCH as a formal and descriptive language.* Formal Language Description Languages for Computer Programming, T. B. Steel, ed., North-Holland, Amsterdam, pp. 179–197.

——— (1975), *λ-Calculus and computer science theory*, Proc. Rome Symp., March 1975, Lecture Notes in Computer Science, vol. 37, Springer-Verlag, Berlin.

R. M. BURSTALL (1972a), *Some techniques for proving the correctness of programs which alter data structures*, Machine Intelligence vol. 7, B. Meltzer and D. Michie, eds., Edinburgh, pp. 23–50.

——— (1969), *Proving properties of programs by structural induction*, Comput. J., 12, pp. 41–48.

——— (1972b), *An algebraic description of programs with assertions, verification and simulation*, Proc. ACM Conf. on Proving Assertions about Programs, Las Cruces, New Mexico, pp. 7–14.

R. BURSTALL AND J. W. THATCHER (1974), *The algebraic theory of recursive program schemes*, Category Theory Applied to Computation and Control, E. Manes, ed., Univ. of Mass., pp. 154–160.

J. M. CADIOU (1973), *Recursive definitions of partial functions and their computations*, Ph.D. thesis, Computer Sci. Dept., Stanford Univ., Stanford, Calif.

J. M. CADIOU AND J. J. LEVY (1973), *Mechanizable proofs about parallel processes*, Presented at the 14th Ann. Symp. on Switching and Automata Theory.

J. W. CASE (1971), *Enumeration reducibility and the partial degrees*, Ann. Math. Logic, 2, pp. 419–440.

E. ČECH (1966), *Topological Spaces*, Prague.

A. K. CHANDRA (1974), *Degrees of translatability and canonical forms in program schemas part 1*, IBM Res. Rep. RC4733, Yorktown Heights, N.Y.

——— (1974a), *The Power of Parallelism and Nondeterminism in Programming*, IBM Res. Rep. RC4776, Yorktown Heights, N.Y.

——— (1974b), *Generalized program schemas*, IBM Res. Rep. RC4827, Yorktown Heights, N.Y.

A. K. CHANDRA AND Z. MANNA (1973), *On the power of programming features*, Artificial Intelligence Memo. AIM-185, Stanford Univ., Stanford, Calif.

A. CHURCH (1951), *The calculi of lambda-conversion*, Annals of Mathematical Studies, vol. 6, Princeton University Press, Princeton, N.J.

M. J. CLINT (1972), *Program proving: Co-routines*, Acta Informatica, 2, pp. 50–63.

R. C. CONSTABLE AND D. GRIES (1972), *On classes of program schemata*, this Journal, 1, pp. 66–118.

D. C. COOPER (1966), *The equivalence of certain computations*, Comput. J., 9, pp. 45–52.

B. COURCELLE, G. KAHN AND J. VUILLEMIN (1974), *Algorithmes d'equivalince pour des equations recursive simples*, Proc. 2nd Colloq. on Automata Languages and Programming, Saarbrucken; Lecture Notes in Computer Science, Springer-Verlag, Berlin, pp. 213–220.

B. COURCELLE AND J. VUILLEMIN (1974), *Completude d'un systeme formel pour des equations recursives simples*, Compte-Rendu du Colloque de Paris.

——— (1974), *Semantics and axiomatics of a simple recursive language*, Rapport Laboria, vol. 60, IRIA: Also: 6th Ann. ACM Symp. on the Theory of Computing, Seattle, Washington.

H. B. CURRY AND R. FEYS (1958), *Combinatory Logic*, vol. 1, North-Holland, Amsterdam.

H. B. CURRY, J. R. HINDLEY AND J. SELDIN (1971), *Combinatory Logic*, vol. 2, North-Holland, Amsterdam.

M. DAVIS (1958), *Computability and Unsolvability*, McGraw-Hill, New York.

E. W. DIJKSTRA (1974), *A simple axiomatic basis for programming language constructs.* Indag. Math., 36, pp. 1–15.
——— (1976), *A Discipline of Programming,* Prentice-Hall, Englewood Cliffs, N.J., to appear.
——— (1972), *Notes on structured programming,* Structural Programming, C. A. R. Hoare, E. W. Dijkstra and O. J. Dahl, Academic Press, New York.
J. DONAHUE (1974), *Scottery Child's Guide No. 1,* Dept. of Comput. Sci., Univ. of Toronto.
B. N. DUONG (1974), *A high level, variable free calculus for recursive programming,* Comput. Sci. Dept. Rep. CS 74 03, Univ. of Waterloo.
S. EILENBERG AND C. C. ELGOT (1970), *Recursiveness,* Academic Press, New York.
——— (1974), *Automata, Languages and Machines,* vol. A, Academic Press, New York (vol. B in press).
C. C. ELGOT (1971), *Algebraic theories and program schemes,* Symposium on Semantics of Algorithmic Languages, E. Engeler, ed., Lecture Notes in Mathematics, vol. 188, Springer-Verlag, New York.
——— (1967), *Abstract algorithms and diagram closure,* IBM Res. Rep. RC 1750, Yorktown Heights, N.Y.
C. C. ELGOT, A. ROBINSON AND J. D. RUTLEDGE (1966), *Multiple control computer models,* IBM Res. Rep. RC 1622, Yorktown Heights, N.Y.
M. H. VAN EMDEN AND R. A. KOWALSKI (1974), *The semantics of predicate logic as a programming language,* Memo. no. 83 (MIP-R-103), Dept. of Machine Intelligence, Univ. of Edinburgh.
E. ENGELER (1968), *Formal languages: Automata and structures,* Lectures in Advanced Mathematics, Markham.
——— (1974), *Algorithmic Logic, Foundations of Computer Science,* J. W. de Bakker, ed., Mathematical Centre Tract MCT 63, Amsterdam.
——— (1967), *Algorithmic properties of structures,* Mathematical Systems Theory, 1, pp. 183–195.
J. ENGELFREIT (1974), *Simple program schemes and formal languages,* Springer Lecture Notes in Computer Science, vol. 20.
J. ENGELFREIT AND E. M. SCHMIDT (1975), *IO and OI,* DAIMI PB-47, Matematisk Institut, Aarhus Universitet, Datalogisk Afdeling, Denmark.
JN. L. ERSHOV (1971), *Computable numerations of morphisms,* Algebra and Logic, 10, pp. 247–308.
——— (1972a), *Computable functions of finite types,* Ibid., 11, pp. 367–437.
——— (1972b), *Everywhere defined (total) continuous functionals,* Algebra and Logic, 11, pp. 656–665.
——— (1972d), *Continuous lattices and A-spaces,* Dockl. Acad. Nauk USSR, 207, pp. 523–526.
——— (1972e), *Theory of A-spaces,* Algebra and Logic, 12, pp. 369–916.
——— (1973), *Theorie der Numerierungen,* Veb. Deutscher Verlag der Wissenschaften, Berlin.
A. E. FISCHER AND M. J. FISCHER (1973), *Mode modules as representations of domains,* Proc. ACM Symp. on Principles of Programming Languages, Boston, pp. 139–143.
M. J. FISCHER (1972), *Lambda calculus schemata,* Proc. ACM Conf. Proving Assertions about Programs, Las Cruces, N.M., pp. 104–109.
R. W. FLOYD (1967), *Assigning meanings to programs,* Proc. Symp. in Appl. Math., vol. 19, Mathematical Aspects of Computer Science, J. T. Schwartz, ed., pp. 33–41.
M. M. FOKKINGS (1974), *Inductive Assertion Patterns for Recursive Procedures,* Compte-Rendu de Colloque de Paris.
R. M. FRIEDBERG AND H. R. ROGERS (1959), *Reducibility and completeness for sets of integers,* Z. Math. Logik Grunlagen Math., 5, pp. 117–125.
H. FRIEDMAN (1971), *Algorithmic procedures, generalized Turing algorithms, and elementary recursion theory,* Logic Colloquium 1969, North-Holland, Amsterdam, pp. 361–389.
S. J. GARLAND AND D. C. LUCKHAM (1972), *Translating recursion schemes into program schemes,* Proc. ACM Conf. on Proving Assertions about Programs, Las Cruces, N.M., pp. 83–96.
H. GERICKE (1966), *Lattice Theory,* Harrap.
J. A. GOGUEN (1967), *L-fuzzy sets,* J. Math. Anal. Appl., 18, pp. 145–174.
——— (1969), *Categories of L-sets,* Bull. Amer. Math. Soc., 75, pp. 524–622.
——— (1972), *On homomorphisms, simulations, correctness, subroutines and termination for programs and program schemes,* Proc. 13th IEEE Conf. on Switching and Automata Theory, pp. 52–60.

——— (1974), *Semantics of computation*, Category Theory Applied to Computation and Control, E. Manes, ed., Univ. of Mass., pp. 234–249.

J. A. GOGUEN ET AL. (1975), *Initial algebra semantics*, IBM Res. Rep. RC 5243, Yorktown Heights, N.Y.

M. J. C. GORDON (1973a), *Models of pure LISP*, Experimental Programming Rep. 31, School of Artificial Intelligence, Univ. of Edinburgh. (Also Ph.D. thesis, *Evaluation and denotation of pure LISP programs: A worked example in semantics.*)

——— (1973b), *An extended abstract of "Models of pure LISP"*, Memo. SAI-RM-7, School of Artificial Intelligence, Univ. of Edinburgh.

——— (1975a), *Operational Reasoning and denotational semantics*, Artificial Intelligence Memo. 264, Stanford Univ., Stanford, Calif.

——— (1975b), *Towards a semantic theory of dynamic binding*, Artificial Intelligence Memo. 265, Stanford Univ., Stanford, Calif.

G. GRATZER (1968), *Universal Algebra*, Van Nostrand, New York.

I. GUESSARIAN (1974), *Sur une reduction des schemas de programmes polyadiques a des schemas monadiques et ses applications*, Compte-Rendu de Colloque de Paris.

L. GUTTERIDGE (1971), *Some Results on Enumeration Reducibility*, Ph.D. dissertation, Simon Fraser Univ., Burnaby, B.C., Canada.

J. R. HINDLEY, B. LERCHER AND J. P. SELDIN (1972), *Introduction to combinatory logic*, London Mathematical Society Lecture Note Series, vol. 7, Cambridge University Press, Cambridge, England.

J. R. HINDLEY AND G. MITSCHKE (1975), *Some remarks about the connections between combinatory logic and axiomatic recursion theory*, Preprint 203, Fachbereich Mathematik, Technische Hochschule Darmstadt.

P. HITCHCOCK (1974), *An approach to formal reasoning about programs*, Ph.D. thesis, Univ. of Warwick, England.

P. HITCHCOCK AND D. M. R. PARK (1972), *Induction rules and proofs of termination*, Proc. Colloques IRIA, Theorie des Automates des Languages et de la Programmation.

C. A. R. HOARE (1969), *An axiomatic basis of computer programming*, Comm. ACM, 12, pp. 576–580, 583.

——— (1971a), *Proof of a program: FIND*, Ibid., 14, pp. 39–45.

——— (1971b), *Procedures and parameters: An axiomatic approach*, Symposium on Semantics of Programming Languages, E. Engeler, ed., Lecture Notes in Mathematics, vol. 188, Springer-Verlag, New York, pp. 102–116.

——— (1972), *Notes on data structuring*, Structured Programming, C. A. R. Hoare, E. W. Dijkstra and O. J. Dahl, Academic Press, New York.

C. A. R. HOARE AND P. LAUER (1974), *Consistent and complementary formal theories of the semantics of programming languages*, Acta Informatica, 3, pp. 135–153.

C. A. R. HOARE AND N. WIRTH (1972), *An axiomatic definition of the programming language PASCAL*, Bericht der Fachgrüppe Computer-Wissenschaften 6, Eidenossische Technische Höchschule, Zürich.

C. HOSONO AND M. SATO (1975), *A solution to Scott's problem: "Do the retracts in $P\omega$ form a continuous lattice?"*, Research Inst. for Mathematical Sci., Kyoto (preprint).

G. HOTZ (1966), *Eindeutigkeit und Mehrdeutigkeit formaler Sprachen, Electron. Informationsverarbeit*, Kybernetik (Berlin), 2, pp. 235–246.

J. M. E. HYLAND (1975a), *Recursion theory on the countable functionals*, D. Phil. thesis, Oxford Univ., Oxford, England.

——— (1975b), *A survey of some useful partial order relations in terms of the lambda calculus*, Proc. Conf. on λ-Calculus and Computer Science Theory, Rome, pp. 83–95.

——— (to appear), *A syntactic characterization of the equality in some models for the lambda calculus*, J. London Math. Soc.

Y. I. IANOV (1970), *The logical scheme of algorithms*, Problems in Cybernetics, vol. 1, Pergamon Press, New York.

S. IGARASHI (1972), *Admissibility of fixed point induction in first order logic of typed theories*, Artificial Intelligence Memo. AIM-168, Computer Sci. Dept., Stanford Univ., Stanford, Calif.

K. INDERMARK (1974), *Gleichungsdefinierbarkeit in Relationalstukturen*, Habilitationsschrift, Mathematisch Naturwissen-schaftlichen Fakultat der Rheinischen Freidrich-Wilhelms-Universatat, Bonn, W. Germany.

C. B. JONES (1974), *Mathematical semantics of goto: Exit formulation and its relation to continuations*, preprint.

G. KAHN (1973), *A preliminary theory of parallel programs*. Rapport Laboria IRIA.

D. M. KAPLAN (1969), *Regular expressions and the equivalence of programs*, J. Comput. System Sci., 3, pp. 361–385.

R. M. KARP AND R. E. MILLER (1969), *Parallel program schemata*, Ibid., 3, pp. 147–195.

D. J. KFOURY (1972), *Comparing algebraic structures up to algorithmic equivalence*, Automata, Languages and Programming, M. Nivat, ed., North-Holland, Amsterdam, pp. 253–263.

——— (1974), *Translatability of schemas over restricted interpretations*, J. Comput. System Sci., 8, pp. 387–408.

S. C. KLEENE (1950), *Introduction to Metamathematics*, Van Nostrand, New York.

B. KNASTER (1928), *Un theorème sur les fonctions d'ensembles*, Ann. Soc. Polon. de Math., 8, pp. 133–134.

P. J. LANDIN (1964), *The mechanical evaluation of expressions*, Comput. J. 6, pp. 308–320.

——— (1965), *A correspondence between ALGOL 60 and Church's lambda notation*, Comm. ACM, 8, pp. 89–101.

——— (1966a), *A λ-Calculus approach*, Advances in Programming and Non-numerical Computation, L. Fox, ed., Pergamon Press, New York, pp. 97–141.

——— (1966b), *The next 700 programming languages*, Comm. ACM, 9, pp. 157–164.

——— (1966a), *A formal description of ALGOL 60*, Formal Language Description Languages for Computer Programming, T. B. Steel, ed., North-Holland, Amsterdam, pp. 266–294.

——— (1969), *A program/machine symmetric automata theory*, Machine Intelligence, vol. 5, B. Meltzer and D. Michie, eds., Edinburgh University Press, pp. 99–120.

P. J. LANDIN AND R. M. BURSTALL (1969), *Programs and their proofs: An algebraic approach*, Machine Intelligence, vol. 4, American Elsevier, New York, pp. 17–44.

J. LESZCZYKOWSKI (1971), *A theorem on resolving equations in the space of languages*, Bull. Acad. Polon. Sci. Sér. Sci. Math. Astronom. Phys., 19, pp. 967–970.

C. H. LEWIS AND B. K. ROSEN (1973–74), *Recursively defined data types, Parts I and II*. IBM Res. Reps. RC 4429 (1973), RC 4713 (1974), Yorktown Heights, N.Y.

B. LISKOV AND S. ZILLES (1973), *An approach to abstraction computation structures*, Group Memo. 88, Project MAC, Mass. Inst. of Tech., Cambridge, Mass.

D. LUCKHAM, D. M. R. PARK AND M. PATERSON (1970), *On formalized computer programs*, J. Comput. System Sci., 4, pp. 220–249.

S. MACLANE (1972), *Categories for the Working Mathematician*, Springer-Verlag, New York.

E. MANES (1974), *Category Theory Applied to Computation and Control*, Proc. 1st Internat. Symp. Math. Dept. and the Dept. of Computer and Information Sci., Univ. Mass., Amherst. Also: Lecture Notes in Computer Science, vol. 26, Springer-Verlag, New York.

Z. MANNA (1969), *The correctness of programs*, J. Comput. System Sci., 3, pp. 119–127.

——— (1974), *Mathematical Theory of Computation*, McGraw-Hill, New York.

Z. MANNA AND J. M. CADIOU (1972), *Recursive definitions of partial functions and their computations*, Proc. ACM Conf. on Proving Assertions about Programs, Las Cruces, N.M., pp. 58–65.

Z. MANNA AND J. MCCARTHY (1970), *Properties of programs and partial function logic*, Machine Intelligence 5, B. Meltzer and D. Michie, eds., Edinburgh University Press, pp. 27–38.

Z. MANNA, Z. NESS AND J. VUILLEMIN (1972), *Inductive methods for proving properties of programs*, Proc. ACM Conf. on Proving Assertions about Programs, Las Cruces, N.M., pp. 27–50.

Z. MANNA AND A. PNUELI (1970), *Formalization of properties of functional programs*, J. Assoc. Comput. Mach., 17, pp. 555–569.

——— (1972), *Axiomatic approach to total correctness of programs*, Artificial Intelligence Lab. Memo. AIM-210, Stanford Univ., Stanford, Calif.

Z. MANNA AND J. VUILLEMIN (1972), *Fixpoint approach to the theory of computation*, Comm. ACM, 15, pp. 528–536.

G. MARKOWSKY (1974), *Categories of chain-complete posets*, RC 5100, Comput. Sci. Dept. IBM T. J. Watson Research Center, Yorktown Heights, N.Y.

——— (1974), *Chain-complete posets and directed sets with applications*, RC 5024, IBM T. J. Watson Research Center, Yorktown Heights, N.Y.

A. MAZURKIEWICZ (1973), *Proving properties of processes*, PRACE CO PAN-CC PAS Reports, vol. 134, Warsaw.

——— (1971), *Proving algorithms by tail functions*, Working Paper for IFIP WG2.2, Feb. 1970. Since published in: Information and Control, 18 (1971), pp. 220–226.

J. MCCARTHY (1963a), *A basis for a mathematical theory of computation*, Computer Programming and Formal Systems, D. Braffort and D. Hirshberg, eds., North-Holland, Amsterdam, pp. 37–70.

——— (1962), *The LISP 1.5 Programmers' Manual*, MIT Press, Cambridge, Mass.

——— (1963b), *Towards a mathematical science of computation*, Information Processing 1962, Proc. IFIP Congress 1962, C. M. Poppleworth, ed., North-Holland, Amsterdam, pp. 21–28.

——— (1966), *A formal description of a subset of ALGOL*, Formal Language Description Languages for Computer Programming, R. B. Steel, ed., North-Holland, Amsterdam, pp. 1–12.

J. MCCARTHY AND J. PAINTER (1967), *Correctness of a computer for arithmetic expressions*, Mathematical Aspects of Computer Science, J. T. Schwartz, ed., Proc. of a Symposium in Applied Mathematics, vol. 19, pp. 33–41.

R. E. MILNE (1974), *The formal semantics of computer languages and their implementations*, Ph.D. thesis, Cambridge Univ., Cambridge, England. [Also: Technical Monograph PRG-13, Oxford Univ. Computing Lab., Programming Research Group].

R. MILNER (1972), *Implementation and Application of Scott's Logic for Computable Functions*, Proc. ACM Conf., Las Cruces, N.M., pp. 1–6.

——— (1973), *Models of LCF*, Artificial Intelligence Memo. AIM-186, Computer Sci. Dept., Stanford Univ., Stanford, Calif.

——— (1969), *Program schemes and recursive function theory*, Machine Intelligence 5, B. Meltzer and D. Michie, eds., Edinburgh University Press, pp. 39–58.

——— (1970a), *Algebraic theory of computable polyadic functions*, Computer Sci. Memo., vol. 12, University College, Swansea.

——— (1970b), *Equivalences on program schemes*, J. Comput. System Sci., 4, pp. 205–219.

R. MILNER AND R. WEYHRAUCH (1972), *Proving compiler correctness in a mechanized logic*. Machine Intelligence, 7, pp. 51–72.

R. MILNER (1973a), *An approach to the semantics of parallel programs*, Proc. Convgno di Informatica Teorica, Instituto di Elaborazione delle Informazione, Pisa, Italy.

——— (1973b), *Processes: A mathematical model of computing agents*, Proc. Colloq. in Mathematical Logic, Bristol, England.

F. LOCKWOOD MORRIS (1973), *Advice on structuring compilers and proving them correct*, Proc. SIGACT/SIGPLAN Symp. on Principles of Programming Languages, Boston, pp. 144–152.

——— (1970), *The next 700 programming language descriptions*, Computer Center, Univ. of Essex (typescript).

——— (1972), *Correctness of translations of programming languages*, Computer Sci. Memo. CS 72-303, Stanford Univ., Stanford, Calif.

J. H. MORRIS (1968), *λ-calculus models of programming languages*, Ph.D. thesis, Sloan School of Management, MIT MAC Reprint TR-57, Mass. Inst. of Tech., Cambridge, Mass.

——— (1971), *Another recursion induction principle*, Comm. ACM, 14, pp. 351–354.

——— (1973), *Types are not sets*, Proc. ACM Symp. on Principle of Programming Languages, Boston, pp. 120–124.

——— (1972), *A correctness proof using recursively defined functions*, Formal Semantics of Programming Languages, R. Rustin, ed., Prentice-Hall, Englewood Cliffs, N.J., pp. 107–224.

J. MORRIS AND R. NAKAJIMA (1973), *Mechanical characterisation of the partial order in lattice models of λ-calculus*, Tech. Rep. 18, Univ. of Calif., Berkeley.

P. D. MOSSES (1974), *The mathematical semantics of ALGOL 60*, Tech. Monograph PRG-12, Oxford Univ. Computing Lab., Programming Research Group.

——— (1975a), *The semantics of semantic equations*, Mathematical Foundations of Computer Science, Goos and Hartmanis, eds., Lecture Notes in Computer Science, vol. 28, Springer-Verlag, New York, pp. 409–422.

——— (1975b), *Mathematical semantics and compiler generation*, Thesis, Oxford Univ., Oxford, England.

J. MYHILL AND J. C. SHEPHERDSON (1955), *Effective operations on partial recursive functions*, Z. Math. Logik Grundlagen Math., 1, pp. 310–317.

R. NAKAJIMA (1975), *Infinite normal forms for λ-calculus*, λ-Calculus and Computer Science Theory, C. Böhm, ed., Springer-Verlag, Berlin, pp. 62–82.

A. NERODE (1959), *Some Stone spaces and recursion theory*, Duke Math. J., 26, pp. 397–406.

M. NIVAT (1972), *Languages Algebriques sur le Magma Libre et Semantique des Schemes de Programmes*, Proc. IRIA Symposium on Automata Formal Languages and Programming, North-Holland, Amsterdam.

D. C. OPPEN (1975), *On logic and program verification*, Tech. Rep. 82, Dept. Computer Sci., Univ. of Toronto.

D. M. R. PARK (1970a), *Fixpoint induction and proofs of program properties*, Machine Intelligence, 5, B. Meltzer and D. Michie, eds., American Elsevier, New York, pp. 59–78.

——— (1970b), *Notes on a formalism for reasoning about schemes*, Univ. of Warwick, unpublished notes.

——— (1970c), *The Y combinator Scott's lambda-calculus models*, Univ. of Warwick, unpublished notes.

——— (1970d), *Finiteness is Mu-ineffable*, Theory of Computation Report, No. 3, Dept. of Computer Sci., Univ. of Warwick (1974).

M. S. PATERSON (1963), *Program schemata*, Machine Intelligence, 3, D. Michie, ed., American Elsevier, New York, pp. 19–32.

M. S. PATERSON AND C. S. HEWITT (1970), *Comparative schematology*. Record of Project MAC Conf. on Concurrent Systems and Parallel Computation, Association for Computing Machinery, New York, pp. 119ff.

R. PLATEK (1964), *New foundations for recursion theory*, Thesis, Stanford Univ., Stanford, Calif., unpublished.

G. D. PLOTKIN (1972), *A set-theoretical definition of application*, Memo. MIP-R-95, School of Artificial Intelligence, University of Edinburgh.

——— (1973a), *The λ-calculus is ω-incomplete*, Res. Memo. SAI-RM-2, School of Artificial Intelligence, Univ. of Edinburgh.

——— (1973b), *Lambda-definability and logical relations*. Memo. SAI-RM-4, School of Artificial Intelligence, Univ. of Edinburgh.

——— (1973c), *Call by name, Call-by-value and the λ-calculus, I*, Res. Memo. SAI-RM-6, School of Artificial Intelligence, Univ. of Edinburgh.

——— (1975), *A powerdomain construction*, Dept. of Artificial Intelligence Res. Rep. 3, Univ. of Edinburgh.

H. RASIOWA (1973), *On ω^+-valued algorithmic logic and related problems*, Supplement to Proc. Symp. and Summer School on Mathematical Foundations of Computer Sci., High Tatras, Czechoslovakia.

P. RAULEFS (1975a), *The overtyped lambda calculus*, Interner Bericht Nr. 2, Institut für Informatik, Universität Karlsruhe.

——— (1975b), *Standard models of the overtyped lambda-calculus*, Interner Bericht Nr. 3, Institut für Informatik, Universitat Karlsruhe.

R. R. REDIEJOWSKI (1972), *The theory of general events and its application to parallel programming*, T.P. 18.220 IBM Nordic Laboratory, Sweden.

J. C. REYNOLDS (1969), *Gedanken: A Simple Typeless Language*, Reprint 7621, Argonne National Laboratory, Argonne, Ill.

——— (1972a), *Definitional interpreters for higher order programming languages*, Proc. ACM 25th Nat. Conf., Boston, vol. 2, pp. 717–740.

——— (1972b), *Notes on a lattice-theoretic approach to the theory of computation*, Systems and Information Sci. Dept. Syracuse Univ., Syracuse, N.Y.

——— (to appear), *Towards a theory of type structure*, Colloq. on Programming Paris, April 1974, Lecture Notes in Computer Science, Springer-Verlag, Berlin.

——— (1974a), *On the relation between direct and continuation semantics*, Proc. 2nd Colloq. on Automata, Languages and Programming, Saarbrucken, Lecture Notes in Computer Science, vol. 14, Springer-Verlag, Berlin, pp. 141–156.

——— (1974b), *Semantics of the lattice of flow diagrams*, Systems and Information Science Dept., Syracuse Univ., Syracuse, N.Y. (preprint).

J. C. REYNOLDS (to appear), *On the interpretations of Scott's domains*, Symposia Mathematica.

W. P. DE ROEVER (1974a), *Recursion and parameter mechanisms: An axiomatic approach*, Proc. 2nd Colloq. on Automata Languages and Programming, Saarbrucken, Lecture Notes in Computer Science, vol. 14, Springer-Verlag, Berlin, pp. 34–65.

——— (1974b), *Operational, mathematical and axiomatized semantics for recursive procedures and data structures*, Mathematical Centre Report ID/1, Amsterdam.

H. R. ROGERS (1967), *Theory of Recursive Functions and Effective Computability*, McGraw-Hill, New York.

B. K. ROSEN (1973), *Tree-manipulating systems and Church–Rosser theorems*, J. Assoc. Comput. Mach., 20.

J. D. RUTLEDGE (1970a), *Program schemata as automata, part 1*, IBM Res. RC 3098; J. Comput and System Sci., 7 (1973), pp. 543–578.

——— (1970b), *Parallel processes · · · Schemata and transformation*, IBM Res. Rep. RC 2912, Yorktown Heights, N.Y.

A. SALWICKI (1970a), *Formalized algorithmic languages*, Bull. Acad. Polon. Sci. Sér. Math. Astronom. Phys., 18, pp. 227–232.

——— (1970b), *On the equivalence of FS-expressions and programs*, Ibid., 18, pp. 275–278.

——— (1970c), *On the predicate calculi with iteration quantifiers*, Ibid., 18, pp. 279–286.

J. G. SANDERSON (1973), *The lambda calculus, lattice theory and reflexive domains*, Mathematical Institute Lecture Notes, Oxford, England.

L. P. SASSO (1971), *Degrees of unsolvability of partial functions*, Ph.D. dissertation, Univ. of Calif., Berkeley.

——— (1975), *A survey of partial degrees*, J. Symbolic Logic, 40, pp. 130–140.

——— (1973), *A minimal partial degree*, Proc. Amer. Math. Soc., 38, pp. 388–392.

D. SCOTT (1969), *A type-theoretical alternative to ISWIM, CUCH, OWHY*, Unpublished notes, Oxford, England.

——— (1970), *Outline of a mathematical theory of computation*, Proc. 4th Ann. Princeton Conf. on Information Sciences and Systems, pp. 169–176. [Also: Tech. Monograph PRG-2, Oxford Univ. Computing Lab., Programming Research Group (1970).]

——— (1971), *The lattice of flow diagrams*, Symposium on Semantics of Algorithmic Languages, E. Engeler, ed., Lecture Notes in Mathematics, vol. 188, Springer-Verlag, New York, pp. 311–366. [Also: Tech. Monograph PRG-3, Oxford Univ. Computing Lab., Programming Research Group (1970)].

——— (1972a), *Continuous lattices*, Proc. 1971 Dalhousie Conference, Lecture Notes in Mathematics, vol. 274, Springer-Verlag, New York, pp. 97–136. [Also: Tech. Monograph PRG-7, Oxford Univ. Computing Lab., Programming Research Group (1971).]

——— (1972b), *Mathematical concepts in programming language semantics*, AFIPS Conf. Proc., vol. 40, pp. 225–234.

——— (1973a), *Data types as lattices*, Unpublished lecture notes, Mathematical Centre, Amsterdam, 1973.

——— (1973b), *Models for various type-free calculi*, Proc. IVth Internat. Cong. for Logic, Methodology and the Philosophy of Science, Bucharest, P. Suppes et. al., eds., North-Holland, Amsterdam, pp. 157–187.

——— (1975a), *Lambda calculus and recursion theory*, Proc. 3rd Scandinavian Logic Symposium, Stig Kanger, ed., North-Holland, Amsterdam, pp. 154–193.

——— (1975b), *Combinators and classes, λ-Calculus and computer science theory*, Proc. Rome Symp., March 1975; Lecture Notes in Computer Science, vol. 37, Springer-Verlag, Berlin, pp. 1–26.

——— (1975c), *Some philosophical issues concerning theories of combinators, λ-Calculus and computer science theory*, Proc. Rome Symp., March 1975; Lecture Notes in Computer Science, vol. 37, Springer-Verlag, Berlin, pp. 346–366.

D. SCOTT AND C. STRACHEY (1971), *Toward a mathematical semantics for computer languages*, Proc. Symposium on Computers and Automata, Polytechnic Inst. of Brooklyn, vol. 21, pp. 19–46. [Also: Technical Monograph PRG-6, Oxford Univ. Computing Lab., Programming Research Group (1971).]

J. C. SHEPERDSON (1975), *Computation over abstract structures: Serial and parallel procedures and Friedman's effective definitional schemes*, Logic Colloquium 1973, H. E. Rose and J. C. Sheperdson, eds., North-Holland, Amsterdam, pp. 445–513.

C. STRACHEY (1966), *Towards a formal semantics*, Formal Language Description Languages for Computer Programming, T. B. Steel, ed., North-Holland, Amsterdam, pp. 198–220.

——— (1967), *Fundamental concepts in programming languages*, Unpublished lecture notes for the NATO Summer School, Copenhagen.

——— (1972), *Varieties of programming languages*, Proc. Internat. Computing Symp., Cini Foundation, Venice, pp. 222–233. [Also: Tech. Monograph PRG-10, Oxford Univ. Computing Lab., Programming Research Group (1972).]

C. STRACHEY AND C. WADSWORTH (1974), *Continuations: A mathematical semantics for handling full jumps*, Tech. Monograph PRG-11, Oxford Univ. Computing Lab., Programming Research Group.

A. TANG (1974), *Recursion theory and descriptive set theory in effectively given T_0 spaces*, Ph.D. thesis, Princeton Univ., Princeton, N.J.

——— (1975a), *Borel sets in $P\omega$*, IRIA-Laboria, preprint.

——— (1975b), *Notes on subsets on $P\omega$ with extra-finitary property*, IRIA, preprint.

——— (1975c), *Sets of the form $\{x|R(x) = T\}$ in $P\omega$*, IRIA, preprint.

——— (1975d), *$\underset{e}{\leqq}$ Degrees in P*, IRIA, preprint.

——— (1975e), *A hierarchy of B_δ sets in $P\omega$*, IRIA, preprint.

A. TARSKI (1930), *On some Fundamental Concepts of Mathematics* (1930); translated by J. H. Woodger in Logic, Semantics, Metamathematics, Cambridge, 1956, pp. 30–37.

——— (1955), *A lattice-theoretical fixpoint theorem and its applications*, Pacific J. Math., 5, pp. 285–309.

R. D. TENNENT (1973), *Mathematical semantics and design of programming languages*, Ph.D. thesis, Univ. of Toronto. [Also: Tech. Rep. 59, Univ. of Toronto (1973).]

——— (1974a), *The mathematical semantics of programming languages*, Dept. of Computing and Information Sci., Queen's Univ., Kingston, Ontario, Canada (preprint).

——— (1974b), *A contribution to the development of PASCAL-like languages*, Tech. Rep. 74-25, Dept. of Computing and Information Sci., Queen's Univ., Kingston, Ontario, Canada.

——— (1975), *PASQUAL: A proposed generalization of PASCAL*, Tech. Rep. 75-32, Dept. of Computing and Information Sci., Queen's Univ., Kingston, Ontario, Canada.

J. VUILLEMIN (1973a), *Proof techniques for recursive programs*, IRIA—Laboria Rep.

——— (1973b), *Correct and optimal implementations of recursion in a simple programming language*, IRIA—Laboria Rep. 24.

C. P. WADSWORTH (1971), *Semantics and pragmatices of the lambda-calculus*, D. Phil. thesis, Oxford Univ., Oxford, England.

——— (1975a), *The relation between lambda-expressions and their denotation*, Dept. of Systems and Information Sci., Syracuse Univ., Syracuse, N.Y. (preprint).

——— (1975b), *Approximate reduction and lambda-calculus models*, Dept. of Systems and Information Sci., Syracuse Univ., Syracuse, N.Y.

——— (1975c), Some unusual λ-calculus numeral systems, in preparation.

——— (to appear), *On the topological ordering in the D-model of the lambda-calculus*, in preparation.

E. G. WAGNER (1971a), *Languages for defining sets in arbitrary algebras*, Proc. of the 11th IEEE Conf. on Switching and Automata Theory, pp. 192–201.

——— (1971b), *An algebraic theory of recursive definitions and recursive languages*, Proc. of the 3rd ACM Symp. on Theory of Computing.

——— (1974), *Notes on categories, algebras and programming languages*, Unpublished lecture notes, London.

M. WAND (1973), *Mathematical foundations of formal language theory*, MAC-TR-108, Project MAC, Mass. Inst. of Tech., Cambridge, Mass.

——— (1974a), *An algebraic formulation of the Chomsky hierarchy*, Category Theory Applied to Computation and Control, E. Manes, ed., Univ. of Mass., pp. 216–221.

——— (1974b), *On the recursive specification of data types*, Category Theory Applied to Computation and Control, E. Manes, ed., Univ. of Mass., pp. 222–225.

——— (1974c), *Realizing data structures as lattices*, Tech. Rep. 11, Computer Sci. Dept., Indiana Univ.

——— (1975), *Fixed-point constructions in order-enriched categories*, Tech. Rep. 23, Computer Sci. Dept., Indiana Univ.

P. WEGNER AND D. LEHMANN (1972), *Algebraic and topological prerequisites to Scott's theory of computation*, Tech. Rep. 72-2, Dept. of Computer Sci., Hebrew Univ. of Jerusalem.

R. W. WEYRAUCH AND R. MILNER (1972), *Program correctness in a mechanized logic.* Proc. of the 1st USA–JAPAN Computer Conf., pp. 384–390.

J. B. WRIGHT (1972), *Characterization of recursively enumerable sets*, J. Symbolic Logic, 37, pp. 507–511.