OUTLINE OF A MATHEMATICAL THEORY OF COMPUTATION

by

Dana Scott

Oxford University Computing Laboratory

Programming Research Group

OUTLINE OF A MATHEMATICAL THEORY OF COMPUTATION

by

Dana Scott
Princeton University

Technical Monograph PRG-2
November 1970

Oxford University Computing Laboratory,
Programming Research Group,
45 Banbury Road,
Oxford.

Department of Philosophy,
1879 Hall,
Princeton University,
Princeton, New Jersey, 08540.

This paper is a revised and slightly expanded version of a paper under the same title in the Proceedings of the Fourth Annual Princeton Conference on Information Sciences and Systems (1970).

ABSTRACT

This paper introduces a theory of computation which is *mathematical* rather than *operational* in its approach. The theory is based on the idea that data types can be partially ordered by a relation similar to that of approximation, and as a result can be considered as complete lattices. The properties of these lattices and the functions on them are discussed in an informal manner and shown to be very suitable for developing a mathematical theory of the semantics of programming languages. A preliminary result of this approach is the construction of the first "mathematical" model for the *λ-calculus*.

CONTENTS

OUTLINE OF A MATHEMATICAL THEORY OF COMPUTATION

0. INTRODUCTION

The motivation for trying to formulate a mathematical theory of computation is to give *mathematical semantics* for high-level computer languages. The word "mathematical" is to be contrasted in this context with some such term as "operational". Thus the mathematical meaning of a *procedure* ought to be the *function* from elements of the data type of the input variables to elements of the data type of the output. On the other hand, the operational meaning will generally provide a *trace* of the whole *history* of the computation following the sequencing stipulated in the stated procedure definition and will involve an explicit finitary choice of *representations* of data eventually in something close to bit patterns. The point is that, mathematically speaking, functions are independent of their means of computation and hence are "simpler" than the explicitly generated, step-by-step evolved sequences of operations on representations. In giving precise definitions of operational semantics there are always to be made more or less arbitrary choices of schemes for cataloging partial results and the links between phases of the calculation (cf. the formal definitions of such languages as PL/I and ALGOL 68) and to a great extent these choices are irrelevant for a true "understanding" of a program. Mathematical semantics tries to avoid these irrelevancies and should be more suitable to a study of such problems as the *equivalence* of programs.

It is all very well to aim for a more "abstract" and a "cleaner" approach to semantics, but if the plan is to be any good, the operational aspects cannot be completely ignored. The reason is obvious: in the end the program still must be run on a machine - a machine which does not possess the benefit of "abstract" human understanding, a machine that must operate with finite configurations. Therefore, a mathematical semantics, which will represent the first major segment of the complete, rigorous definition of a programming language, must lead naturally to an operational simulation of the abstract entities, which (if done properly) will establish the practicality of the language, and which is necessary for a full presentation.

Thinking only of functions for the moment, it is clear that a mathematically defined function can be known to be comput*able* without its being quite obvious *how* to compute the function in a practical sense - just as it is possible to know that an infinite series is convergent without having a clear idea of its sum. Even though the abstract definition of the function is sufficient to determine it, we cannot really say that the function is *known* until the algorithm is revealed. (Even then our knowledge is somewhat "indirect" or "potential", but never mind.) The conclusion is, then, that an *adequate* theory of computation must not only provide the abstractions (what is computable) but also their "physical" realizations (how to compute them).

What is new in the present theory is exactly these *abstractions*; whereas the means of realization, the techniques of implementation, have been known for some time, as the many, highly complex compilers that are presently in operation demonstrate. Of course, new concepts may require (or suggest) new methods of implementation, but that remains to be seen. However, notice this essential point: unless there is a prior, generally accepted mathematical definition of a language at hand, who is to say whether a proposed implementation is *correct*? What is the "standard" against which a realization is to be measured? Now it is often suggested that the meaning of the language resides in one particular compiler for it. But that idea is wrong: the "same" language can have many "different" compilers. The person who wrote one of these compilers obviously had a (hopefully) clear understanding of the language to guide him, and it is the purpose of mathematical semantics to make this understanding "visible". This visibility is to be achieved by abstracting the central ideas into mathematical entities, which can then be "manipulated" in the familiar mathematical manner. Even if the compiler-oriented approach (even compiled to run on an "abstract" machine) were transparent — which it is not — there would still be interest in bringing out the abstractions to connect the theory with standard mathematical practice.

1. THE PROBLEM OF SELF-APPLICATION

Having this obviously desirable mathematical theory seems to require some new structural notions, some new insights into the nature of *data types* and the *functions* (mappings) that are to be allowed from one to another. Moreover, it soon becomes clear in thinking about "higher-type" programming concepts (e.g. procedures) that spaces of functions must also be considered as forming data types. Since a function (say, mapping integers to integers) is generally in itself an infinite object, it also becomes necessary to introduce some idea of finite *approximation* - just as we do in a sense for real numbers. On top of this there are already operationally "defined" concepts of function which seem to have *no* mathematical counterparts. In particular it is not unknown in programming languages to allow *unrestricted* procedures which can very well produce unrestricted procedures as values. Speaking mathematically this is tantamount to allowing a function that is to be well defined on *all* allowable functions as arguments - a kind of super-functional - and which is even applicable to *itself* as an argument. To date no mathematical theory of functions has ever been able to supply conveniently such a free-wheeling notion of function except at the cost of being inconsistent. The main *mathematical* novelty of the present study is the creation of the proper mathematical theory of functions which accomplishes these aims (consistently!) and which can be used as the basis for the *meta-mathematical* project of providing the "correct" approach to semantics.

It should be stressed at once that the problem of *self-application* arises in ways more crucial to the interpretation of programming languages than in the contemplation of the (to some, impractical) unrestricted procedures. The problem concerns the related questions of keeping track of *side effects* and of the *storage of commands*. In the first place, what is a store? Physically, we have several remarkable answers, but mathematically it comes down to being simply a *mapping* (a function) which connects *contents* to *locations*. Speaking more precisely, the (current) *state* of the store, call it σ, is mathematically a function:

$$\sigma : L \to V$$

which assigns to each location $\ell \in L$ (the set of all locations) its (current) contents $\sigma(\ell) \in V$ (the set of all allowable values). Let S be the set of all states. What is a *side effect*? Obviously a *change* of state. What is a *command*? A *request* for a side effect. More mathematically, a command is a function

$$\gamma : S \to S$$

which transforms (old) states into (new) states.

Question: can a command be stored? Answer: well, we do it *operationally* all the time. Question: is that mathematically justified? Let's see. Suppose σ is the current state of the store, and suppose $\ell \in L$ is a location at which a command is stored. Then $\sigma(\ell)$ is a command; that is

$$\sigma(\ell) : S \to S.$$

Hence, $\sigma(\ell)(\sigma)$ is well defined. Or is it? This is just an insignificant step away from the self-application problem $p(p)$ for "unrestricted" procedures p, and it is just as hard to justify mathematically. Of course, in the operational approach we do not store the command itself as a function but rather a "code word" or "piece of text" that stands for the command in an unambiguous way. But to carry out the formal description of how this works – especially for compound commands depending on parameters – involves us in most of the nasty questions of programming language semantics and is not really a satisfactory conceptual way out.

2. DATA TYPES AND MAPPINGS

Getting down to particulars, we must ask: what exactly is a *data type?* To simplify matters, we can identify a data type with the set D of all objects of that type. But this is in itself too simple: the objects *are* structured and bear certain relations to one another, so the type is something more than a set. Now this structuring must not be confused with the idea of *data structures* (lists, trees, graphs, etc.); these will come in later. The kind of structure being discussed here is much more primitive and more general and has to do with the basic sense of *approximation*. Suppose x, $y \in D$ are two elements of the data type, then the idea is not immediately to think of them as being completely *separate* entities just because they may be *different*. Instead y, say, may be a *better* version of what x is trying to approximate. In fact, let us write the relationship

$$x \sqsubseteq y$$

to mean intuitively that y is *consistent with* x and is (possibly) *more accurate than* x. For short we can say that x *approximates* y. This intuitively understood relationship exists on most data types naturally, and it is part of the thesis of this paper that a data type should *always* be provided with such a relationship. This may require some adjustment of thought to accommodate certain standard ideas, but it seems worth the effort to unify the treatment of various types.

So let us agree for the sake of argument that types D are structured by relations $\sqsubseteq$ (at least). What can we say abstractly about such a relationship? With reference to the intuitive understanding, it is clear that we want to assume that $\sqsubseteq$ is *reflexive*, *transitive*, and *antisymmetric*.

AXIOM 1. A data type is a partially ordered set.

That may not seem like much (partially ordered sets are so very general) but it is slight progress. The next bit of progress should concern mappings.

Suppose D and D' are two data types (with appropriate partial orderings $\sqsubseteq$ and $\sqsubseteq'$). Suppose $f\colon D \to D'$ is a reasonable mapping of the elements of the one into the other. Should there be anything to say in general about properties of mappings? Well, suppose $x, y \in D$ and $x \sqsubseteq y$. If f were a function defined by a program in any of the usual ways, it would be sensitive to the accuracy of its arguments (inputs) in a special way: *the more accurate the input, the more accurate the output.* In symbols:

$$x \sqsubseteq y \text{ implies } f(x) \sqsubseteq' f(y).$$

In other words, with respect to the partial orderings f is monotonic. We make this an axiom also:

<u>AXIOM 2</u> Mappings between data types are <u>monotonic</u>.

Note that such a condition easily generalizes to functions of several variables, even variables of mixed types.

In numerical computation Axiom 2 is sometimes denied, but this is a confusion about the use of the word *accuracy*. It is true that we know some clever asymptotic algorithms which give *better* answers when the accuracy is *cruder*, but they should be considered as functions of *two* variables: the usual input data together with a parameter indicating the degree of accuracy - or maybe better the number of "terms" to be selected from the "expansion". It can certainly happen that taking more terms just ruins the already good approximation, but note that the input and the number of terms are already supposed known *perfectly*. The notion of accuracy we are trying to capture with the $\sqsubseteq$ relation is something else and does *not* depend on this presupposition. Maybe it would be better to talk about *information*; thus, $x \sqsubseteq y$ means that x and y want to approximate the same entity, but y gives *more* information about it. This means we have to allow "incomplete" entities, like x, containing only "partial" information. (The way to do this in numerical calculation is called *interval analysis*, but we do not have the space here to be more specific.) Allowing for partiality of arguments and values has the good effect that our functions become *partial* too; for even if the arguments are perfect the values may only be partial. This is necessary in considering algorithmically defined functions, since for some combinations of arguments it may happen that the algorithm does not "converge". As a consequence of this point of view, then, there can be no numerical function of the kind allowed by Axiom 2 which maps a "partial" real number to an integer exponent representing the degree of accuracy. But this is not a drawback, as can be seen when one examines the details of the method: there are sufficiently many monotonic functions.

3. COMPLETENESS AND CONTINUITY

The theory based on Axioms 1 and 2 would be too abstract, though it is not vacuous. We need to be more specific about the behaviour of approximations for the applications we have in mind. Thus suppose an infinite sequence of approximations is such that

$$x_0 \sqsubseteq x_1 \sqsubseteq \cdots \sqsubseteq x_n \sqsubseteq x_{n+1} \sqsubseteq \cdots,$$

then it seems reasonable to suppose that the x_n are tending to a *limit*. Call the limit y, and we write

$$y = \bigsqcup_{n=0}^{\infty} x_n,$$

because in the sense of the partial ordering $\sqsubseteq$ the limit is naturally taken to be the *least upper bound* (l.u.b.). If we imagine the successive terms of the sequence as giving us more and more information, than the limit represents a kind of "union" or *join* of the separate contributions. In fact, for mathematical simplicity, we assume that *every* subset of the data type has a least upper bound, which amounts to:

AXIOM 3. A data type is a complete lattice under its partial ordering.

In particular, as is well known, the existence of arbitrary least upper bounds *implies* the existence of greatest lower bounds (g.l.b.), which is why we say we have a complete lattice.

This last assumption requires some explanation. In case $x, y \in D$, it may be that they are either approximations to the same "perfect" entity or not, in which case they are somehow "inconsistent" with one another. In any case we are assuming they have a l.u.b. or *join* $x \sqcup y \in D$. Even worse, we assume that the *whole* of D has a l.u.b., which we call $\top \in D$. This $\top$ is the "top" of the lattice, the very "largest" element in the partial ordering $\sqsubseteq$. It is not to be regarded as a "perfect" element, but rather as an "over-determined" element. Hence, we can regard the equation

$$x \sqcup y = \top$$

as meaning intuitively that x and y are *inconsistent*. This is to be distinguished from the much weaker relationship of being *incomparable*, which is simply: $x \not\sqsubseteq y$ and $y \not\sqsubseteq x$.

The l.u.b. of the *empty* suhset of our data type D is an element $\bot \in D$. This is the "bottom" of the lattice, the very "smallest" element in the partial ordering. It may be regarded as the most "under-determined" element. The l.u.b. of the set of all lower bounds of a subset $X \subseteq D$, is the g.l.b. $\bigsqcap X \in D$. For $x,y \in D$, we have the meet $x \sqcap y = \bigsqcap\{x,y\} \in D$. We can intuitively read the equation

$$x \sqcap y = \bot$$

as meaning that x and y are *unconnected*, in the sense that there is no "overlap" of information between them.

Having supposed that the data type permits "limits", we have to re-examine our view of functions. If a function is *computable* in some intuitive sense, then getting out a "finite" amount of information about one of its values ought to require putting in only a "finite" amount of information about the argument. Now our notion of "information" is *qualitative* rather than *quantitative*, but it is still possible to express this fundamental idea in terms of the notions we have available.

First of all, the proper notion of *limit* is expressed best in terms of *directed sets*. A subset $X \subseteq D$ is *directed* if every finite subset $\{x_0, x_1, \ldots, x_{n-1}\} \subseteq X$ has an upper bound $y \in X$ so that we have:

$$x_0 \sqcup x_1 \ldots \sqcup x_{n-1} \sqsubseteq y.$$

Note that a directed set is always non-empty. (Hint: take $n = 0$.) The *limit* of the directed set is the l.u.b. $\bigsqcup X$. Suppose we want a "finite" amount of information about $\bigsqcup X$. Each "bit" we need (arguing very intuitively) is contained in some element of X; hence by the directedness, *all* of it is contained in at least one single element of X.

Now consider a monotonic function $f: D \to D'$. Given a limit $\bigsqcup X$ of a directed subset $X \subseteq D$, we ask for a "finite" amount of information about $f(\bigsqcup X)$. We agree that to do this requires only a "finite" amount of information about $\bigsqcup X$; thus only a single element $x \in X$ is needed, and it is such that $f(x)$ gives what we want. Note that $f(x) \sqsubseteq f(\bigsqcup X)$ and that $\{f(x) : x \in X\}$ is a directed subset of D'. *All* the information about $f(\bigsqcup X)$ ought to be the "limit" of its "finite" parts. Instead of speaking intuitively and metaphorically, we can express the idea by the *mathematical* equation:

$$f(\bigsqcup X) = \bigsqcup \{f(x) : x \in X\}.$$

In words we can say: the mapping *preserves* the limit. A mapping that preserves *all* limits is called *continuous*. What we have just motivated is:

<u>AXIOM 4</u>. Mappings between data types are <u>continuous</u>.

A function of *two* variables is continuous just when it is continuous in each of its variables separately. Suppose $g(x,y)$ is such a function. It is easy to see that the function f defined by the equation

$$f(x) = g(x,x)$$

is also continuous. (Indeed all the usual methods of *composition* and *identification of variables* produce continuous functions given continuous functions.) The example of f above is the easiest way to obtain a continuous function which does *not* preserve *arbitrary* l.u.b's. (with a suitable choice of g). There is absolutely no reason to suppose that the functions *ought* to preserve g.l.b.'s: one cannot expect any "smoothness" while *decreasing* information. Hence the continuity restriction is about as far as one should go in giving *general* conditions on functions that have a chance of being computable.

4. COMPUTABILITY

So much for the broad outlines of the theory. It still is too abstract, however, because even though certain essential *properties* of computable functions have been isolated, the possibility of "physical" *realization* has not yet been assumed in any form. This we must do. The problem is to restrict attention to exactly those data types where the elements can be approximated by "finite configurations" representable in machines, thereby also making more precise the concept of a "finite amount of information".

The solution to this problem is to take a topological approach: in any case our previous mention of *limits* and *continuity* ought to have suggested that there are some topological ideas in the background. Indeed, any data type D satisfying Axioms 1 and 3 can be regarded as a topological space. To define a topology on a set one needs to say which subsets are *open*. In the case of the data type D, there are two conditions to be satisfied for a subset $U \subseteq D$ to be *open*:

(U_1) whenever $x \in U$ and $x \sqsubseteq y$, then $y \in U$; and

(U_2) whenever $X \subseteq D$ is directed and $\bigsqcup X \in U$, then $X \cap U \neq \phi$.

It is easy to check that D becomes a topological space in this way, and that $f: D \to D'$ is continuous in the *limit preserving* sense if and only if it is continuous in the *topological* sense.

If $x, y \in D$, we write

$$x \prec y$$

to mean that y belongs to the topological *interior* of the upper section determined by x; that is, the interior of the set

$$\{x' \in D: x \sqsubseteq x'\}.$$

The relationship is not as irreflexive as it looks, for there are *isolated* elements x of certain data types such that $x \prec x$. For example $\bot \prec \bot$ always holds; it is probably reasonable to assume $\top$ is isolated also. We write to mean that the g.l.b. of the topological interior of the upper section determined by x is $\sqsubseteq y$. Thus the three relationships

$$x < y,\ x \prec y, \text{ and } x \sqsubseteq y$$

are successively weaker.

Taking the hint from topological spaces like the real numbers (which topologically are a bit different from our data-type spaces), we consider the possibility of having a *dense* subset of the space in terms of which all the other elements can be found as limits. We call such a subset a *basis*. The proper definition seems to be the following: a subset $E \subseteq D$ is a *basis* if it satisfies these two conditions:

(E_1) whenever e, $e' \in E$, then $e \sqcup e' \in E$; and

(E_2) for all $x \in D$ we have $x = \bigsqcup\{e \in E : e < x\}$.

The existence of a basis has several consequences; for example, the meet operation $x \sqcap y$ is continuous if a basis exists, and not necessarily continuous otherwise.

Conditions (E_1) and (E_2) are still not strong enough. make data types "physical". We need the stronger assumption:

<u>AXIOM 5</u>. A data type has an <u>effectively given</u> basis.

That is to say the set E must be "known". Given e, $e' \in E$, we have to know how to find the element $e \sqcup e' \in E$. Given e, $e' \in E$, we have to know how to decide which of the relationships:

$$e < e',\ e \prec e',\ e \sqsubseteq e'$$

are *true* and which are *false*. This certainly is going to require that the set E is at most countahly infinite, and probably that we bave an effective enumeration

$$E = \{e_0, e_1, e_2, \ldots, e_n, \ldots\}$$

in terms of which the above operations and relationships are *recursive* (in the technical sense of recursion theory). To make the data type *really* physical we would need to assume all relationships to be very recursive indeed, but we do not have to go into that here: the intuitive idea of an effectively given basis can be left a bit vague because in particular examples it will be clear what is going on.

Note that as a consequence of Axiom 5 the topologies of data types are *separable* because not only is there a countable dense subset (the basis), but the sets

$$\{x \in D\colon e \prec x\}$$

for $e \in E$ form a countable basis for the topology of D.

The most important consequence of the assumption of an effectively given basis is the possibility of being able to define what it means for an element to be computable. Suppose $x \in D$ and E is the basis. Then (relative to this basis) x is *computable* if and only if there is an effectively given sub-sequence.

$$\{e'_0, e'_1, e'_2, \ldots, e'_n, \ldots\} \subseteq E$$

such that $e'_n \sqsubseteq e'_{n+1}$ for each n, and

$$x = \bigsqcup_{n=0}^{\infty} e'_n.$$

This means that we must be able to give effectively better and better approximations to x which converge to x in the limit. This is an essential notion; because, for one thing, it may very well be the case that the data type D has *uncountably* many elements, while there can be only *countably* many computable elements. Note that there will be in general many sequences converging to an element x, so that just knowing that x is computable does not mean that the "best" way to compute it is also known.

This completes the discussion of the foundations of the subject. Someone may want to point out that Axiom 3 implies Axiom 1 and Axiom 4 implies Axiom 2 — but the axioms are given in the order in which the ideas naturally occur. It can all be said very quickly in summary: data types are complete lattices with effectively given bases and all allowable mappings are to be continuous.

5. CONSTRUCTION OF DATA TYPES

We must now look into the construction of *useful* data types satisfying the axioms, remembering that the lattice structure is only the most primitive structure on a data type, and the "interesting" structure is supplied by various kinds of continuous functions special to the type. (In the examples and constructs to be mentioned, the element $\top$ will always be assumed *isolated*.)

In the first place all *finite* lattices satisfy the axioms, and for them continuity plays no role. Of course finite structures are sufficient for "practical" applications, but many concepts more easily find their expression with reference to infinite structures. In the case of our lattices there are the two numerical data types N and R for the *integers* and the *reals*. As a lattice N has for elements 0, 1, 2, ..., n, ... (these elements are pairwise incomparable under $\sqsubseteq$) plus the two elements $\top$ and $\bot$, respectively above and below all the others (cf. Fig.1). In this case the whole lattice is its own basis.

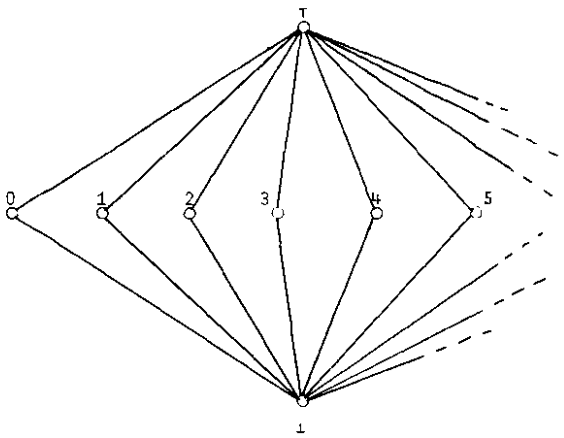

Figure 1.
The Lattice N.

For R the elements are *closed intervals* $[\underline{x},\overline{x}]$ of ordinary real numbers ($\underline{x} \leq \overline{x}$) plus two elements $\top$ and $\bot$. The partial ordering between the intervals is defined thus:

$$[\underline{x},\overline{x}] \sqsubseteq [\underline{y},\overline{y}] \text{ iff } \underline{x} \leq \underline{y} \leq \overline{y} \leq \overline{x}$$

(We use the abbreviation iff in place of the phrase if and only if.) The "perfect" reals are the one-point intervals $[\underline{x},\overline{x}]$ with $\underline{x} = \overline{x}$. The "approximate" reals have $\underline{x} < \overline{x}$. (cf. Fig.2).

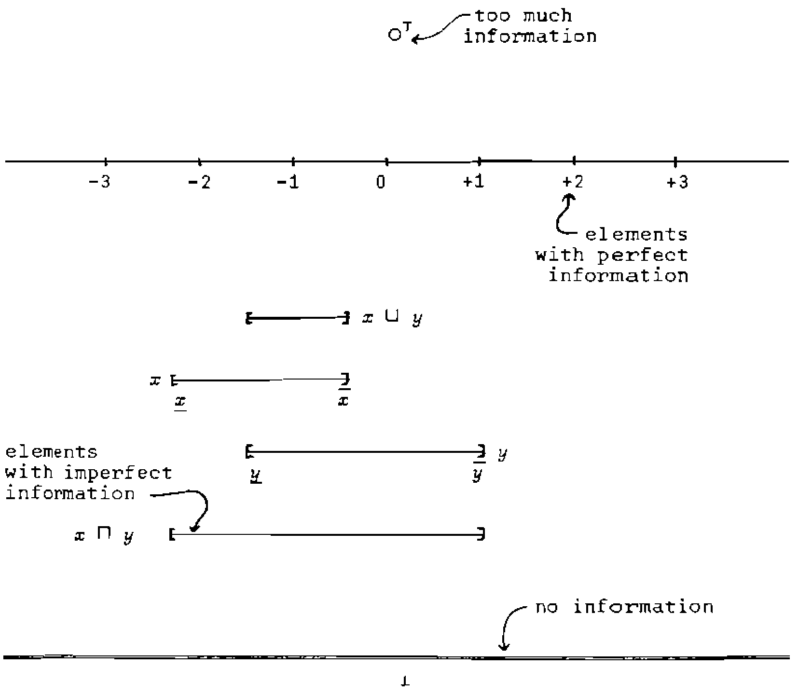

Figure 2

The Lattice R

The basis consists of the intervals with *rational* end points plus the element $\perp$. There is a very close connection between the continuous functions on the lattice R and the ordinary theory of continuous point functions. In both lattices the usual arithmetic operations are represented by continuous functions, and it is especially interesting to consider division in R, as well as other functions with singularities, or functions that are normally left undefined over part of their domain.

Suppose D and D′ are two given data types. There are three particularly important constructs:

$$(D \times D'),\ (D + D'),\ (D \rightarrow D')$$

for obtaining new, "structured" data types from the given ones. The (cartesian) *product* $D \times D'$ has as elements *pairs* $\langle x,x'\rangle$ where $x \in D$ and $x' \in D'$, and for which we define

$$\langle x,x'\rangle \sqsubseteq \langle y,y'\rangle \text{ iff } x \sqsubseteq y \text{ and } x' \sqsubseteq' y'$$

The *sum* is defined as a "disjoint" union of D and D′, except that we identify $\perp = \perp$ and $\top = \top'$. (cf. Fig.3.)

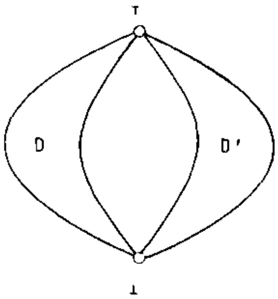

Figure 3.
The Lattice D + D′

The *function space* (D → D′) has as elements all the continuous mappings from D into D′, for which we define

$$f \sqsubseteq g \text{ iff } f(x) \sqsubseteq' g(x) \text{ for all } x \in D.$$

What one must check is how the effectively given bases for D and D′ determine the basis for the construct. This is easy for products and sums but somewhat more trouble for function spaces.

Sums and products can be obviously generalized to more terms and factors, even infinitely many. For example, D^n can be taken as the set of all n-tuples of elements of partially ordered in the obvious coordinate-wise fashion. We can then set

$$D^* = D^0 + D^1 + D^2 + \ldots + D^n + \ldots,$$

which represents the data type of all finite *lists* of elements of the given D. Similarly one can go on to lists of lists of lists of ... . If this were done in the right way, it would seem reasonable that a lattice D^∞ would be obtained such that

$$D^\infty = D + (D^\infty)^*.$$

that is to say, each element of D^∞ is *either* an element of the given D *or* is a list of other elements of D^∞. This sounds very much like the usual kind of recursive definition of lists; but one must take care as the following argument shows.

It is a well-known theorem that every continuous (even monotonic) function mapping a complete lattice into *itself* has a *fixed point*. Applying this remark to the supposed D^∞ above, we note that for fixed $a \in D$, the expression $\langle a,x\rangle$ defines a continuous function of D^∞ into itself. Consider a fixed point:

$$x = \langle a,x\rangle.$$

Thus x is a list whose first term is a and whose second term is . . .? The second term is the list x itself! Thus x is a kind of infinite list:

$$x = \langle a, \langle a, \langle a, \langle a, \ldots \rangle\rangle\rangle\rangle \quad ,$$

which is illustrated in two different ways in Fig.4.

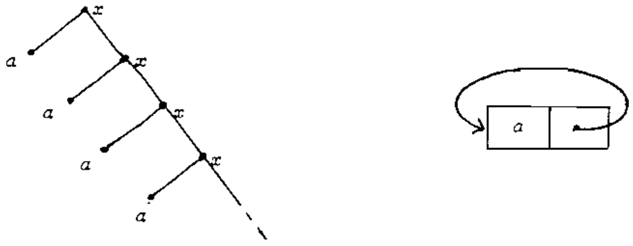

Figure 4
The list $x = \langle a,x \rangle$

That does not square with our usual ideas about data types of lists, but is it bad? The answer is *no*. For it can be shown that the data type D^∞ does in fact exist; it contains all the ordinary finite lists as well as many quite interesting and useful *limits* of sequences of finite lists. One might say that D^∞ gives us the topological *completion* of the space of finite lists, and the various limit points need not be used if one does not care to take advantage of them.

The process of "completing" spaces is a very general one, and the full implications of the method require further exploration. A second example of the completion idea concerns *function spaces*. Let D be given, and set $D_0 = D$ and

$$D_{n+1} = (D_n \to D_n).$$

The spaces D_n are a selection of the "higher-type" spaces of functions of functions of functions of ... . It turns out that there is a natural way of isomorphically embedding each D_n successively into the next space D_{n+1}. These embeddings make it possible to pass to a *limit space* D_∞ which contains the originally given D and is such that

$$D_\infty = (D_\infty \to D_\infty).$$

(Strictly speaking this surprising equation must be taken with a grain of salt: it is only true "up to isomorphism" which is good enough for our purposes.) This space provides the solution to the self-application problem of Section 1, because each element of D_∞ can be regarded as a (continuous!) function on D_∞ into D_∞. And conversely, every continuous function can be represented faithfully by an element. Technically speaking what we have here is the first known, "mathematically" defined model of the so-called *λ-calculus* of Church and Curry.

The reader should take notice of the fact that our abstractly presented theory of computable elements of lattices with effectively given bases applies to the *function* spaces. So we know what computable functions are. Even better we "know" what are the computable elements of the space D_∞ of functions of "infinite" type. Clearly the calculus of operators which can be used to generate computable functions is going to be interesting, and this brings us back to semantics for programming languages. Indeed the natural way to define computable functions is within the context of a suitable programming language. The λ-calculus itself is a programming language: it is the pure language of "unrestricted" procedures. It is only one of the many possible languages.

6. CONCLUSION

We may now sketch the solution to the "storage-of-commands" problem mentioned in Section 1. Let L be the *location* space (finite or, if you like, take L = N, so that the locations are indexed by the integers). The space V of *values* is to be constructed by the limiting methods alluded to above. Supposing it is already constructed, the space S of *states of the store* is defined by:

$$S = (L \to V)$$

The space C of *commands* is defined by:

$$C = (S \to S)$$

The space P of *procedures* — with one parameter and with *side effects* — is defined by:

$$P = (V \to (S \to V \times S)),$$

that is, a procedure is a function, which given first a value of its argument and next given a state of the store, then produces a "computed" value together with the necessary change of the state of the store. Now what can those values be? Well, they might be *numbers* (in N or in R), or they might be *locations*, or they might be *lists*, or they might be *commands*, or they might be *procedures*. (Even to formalize a mode of expression for such an array of possible types of values would already require a fairly involved programming language.) We are thus led to write:

$$V = N + R + L + V^* + C + P$$

If one substitutes the definitions of S, C, and P into this equation, one obtains only a slightly more complicated "definition" than those that we had for D_∞ and D^∞. *Such a space V does exist mathematically*, and it provides the values for expressions of a programming language of the kind we have understood previously in the "operational" way. We should at once begin trying to understand these languages mathematically, because we now have all the tools to do so.

7. BACKGROUND AND REFERENCES

The idea of using monotonic and continuous functions in connection with recursion theory has been current for some time. The author believes, however, that his use of the idea along with more abstract lattices (in particular: with limits of lattices) is new and adds greater flexibility to the program.

The first published references that the author knows are a sequence of four notes by Lacombe in the mid-fifties in which he investigates the notions of recursive real number and recursive function of a real variable. The method is essentially the same as our use of the lattice R sketched above. The exact references are:

Daniel Lacombe, *Extension de la notion de fonction récursive aux functions d'une ou plusieurs variables réeles* I, II, III. Comptes rendus, vol. 240 (1955), pp. 2478-80; vol. 241 (1955), pp. 13-14; vol. 241 (1955), pp. 151-153.

______________, *Remarques sur les opérateurs récursifs et sur les function récursives d'une variable réele.* Ibid, vol. 241 (1955), pp. 1250-52.

The author is indebted to G. Kreisel for pointing out these interesting notes to him. The idea that the method could be extended to the real numbers first became apparent to the author in reading the much later book on numerical analysis:

Ramon E. Moore, *Interval Analysis*, Prentice-Hall (1966), 145 pp.

From a different direction we have the work:

Martin Davis, *Computable functionals of arbitrary finite type*, in Constructivity in Mathematics (A. Heyting, ed.), North Holland (1959), pp. 281-284.

Here the important step is taken of using partial functionals satisfying a consistency condition. Davis relates his ideas to the *continuous* and the *countable* functions of Kreisel and Kleene (cf. their papers in the same volume.) Note, however, that Kleene and Kreisel work generally with *total* functionals, a restriction that is not convenient for the type of applications the author has in mind here.

Further work on partial functionals is contained in:

A. Nerode, *Some Stone spaces and recursion theory*, Duke Mathematical Journal, vol. 26 (1959), pp. 397-406.

Both Nerode and Davis had reported on their ideas at the Cornell Logic Summer School in 1957. A really systematic approach did not appear until:

Richard Platek, *Foundations of Recursion Theory*, Stanford Ph.D. Thesis (1965), unpublished.

It was from Platek in 1963 that the author learned the usefulness of a more abstract approach via monotonic functions. Platek discusses the continuous theory, but not in as much detail as the transfinite recursions. Very recent work on abstract forms of recursion theory has been done by Wagner, Strong, Friedman, and Moschovakis. The explicit connections will have to be discussed elsewhere.

During the spring and summer of 1969 the author worked jointly with J. W. deBakker of Amsterdam on an "algebraic" theory of program schemata and at that time realized that the calculus of monotonic and continuous functions was exactly appropriate to this study. It was a question of David Park about justifying a rule of *double induction* in this calculus together with a conversation with Robin Gandy at Oxford on formulae as operators that brought forward the idea that the calculus should be extended to higher-type function spaces.

In the meantime, during the fall of 1969 at Oxford, the author was working with Christopher Strachey on his theory of semantics of programming languages via L- and R-values and an abstract model of storage. We attempted to apply the formalism of the calculus of recursive functionals of higher types, but were very much hampered by the necessity of having to store *representations* of functions rather than the functions themselves. Having been brought to the right abstract level by the previous work, the author was able to then conjecture that there might very well exist structures with the proper self-referential "storage" properties. And he subsequently was able to offer a rigorous mathematical construction which was finally cast into the realm of lattice theory on account of the simplicity and familiarity of that study. The result was to provide a mathematical basis for the previously employed, entirely formal use of the λ-calculus by Strachey, Landin and others. It was this success that showed that the lattices provided a simple and general enough context to develop a recursion theory involving not only the previously studied structures connected with the integers and the reals, but also some quite new structures. And these new structures seem to be naturally required by programming language semantics besides having mathematical interest. How this connects with McCarthy's program for a mathematical theory of computation will require further discussion in a future publication.