[Sco70]

OUTLINE OF A MATHEMATICAL THEORY OF COMPUTATION

by

Dana Scott
Princeton University

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, rigourous 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 computable 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? 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.

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 take any procedures as arguments and 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 expense of inconsistency. 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 metamathematical 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 an assignment (a function) which connects contents to locations. Speaking more precisely, the (current) state of the store, call it σ, is mathematically a function:

$$\sigma: L \longrightarrow V$$

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 Σ 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

$$\gamma: \Sigma \longrightarrow \Sigma$$

which transforms (old) states to (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,

$$\sigma(\ell): \Sigma \longrightarrow \Sigma$$

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.

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. This intuitively understood relationship exists on most data types naturally, and 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. We can make this into an axiom:

AXIOM 1. A data type is 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 $f: D \to D'$ 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:

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

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

AXIOM 2. Mappings between data types are monotonic.

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.

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 behavior of approximations for the applications we leave 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, then the limit represents a kind of "union" of the separate contributions. In fact, for mathematical simplicity, we assume that every subset of the data type has a least upper bound, which ammounts 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 "largest" element in the partial ordering $\sqsubseteq$. It is not to be considered 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 subset of our data type D is an element $\bot \in D$. This is the "bottom" of the lattice, the "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 reexamine 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 restriction on the functions we are willing to consider: namely, the functions ought to preserve limits.

AXIOM 4. Mappings between data types are continuous.

In full generality we can say this in a precise way in terms of directed sets. A subset $X \subseteq D$ is directed if every finite subset of X has at least one upper bound in the sense of $\sqsubseteq$ in X. (Note: a directed set is nonempty.) A function $f: D \longrightarrow D'$ is continuous iff for all directed subsets $X \subseteq D$ we have:

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

In calculating l.u.b.'s, it is only the l.u.b. of a directed set that ought to be called a limit. There are many examples to show that it would be quite unreasonable to require functions to preserve arbitrary l.u.b.'s. Furthermore 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. Note that the notion of continuity easily extends to accommodate functions of several variables; indeed it turns out that for a function to be continuous in several variables jointly it is sufficient that it be continuous in each of its variables separately.

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 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 $X \cap U \neq \emptyset$.

It is easy to check that D becomes a topological space in this way, and that $f: D \longrightarrow D'$ is continuous in the limit preserving sense iff it is continuous in the topological sense. In the case $x, y \in D$, we write

$$x \prec y$$

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

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

The relationship is not as irreflexive as it looks, for there are isolated elements x of certain data types such that $x \prec x$. We also write

$$x \not\ll y$$

to mean that the g.l.b. of the topological interior of upper section determined by x is $\sqsubseteq y$. Thus the three relationships:

$$x \prec y, \quad x \not\ll y, \text{ and } x \sqsubseteq y$$

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 iff 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 $x = \bigsqcup \{e \in E: e \not\ll x\}$.

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 quite enough to make data types "physical." We need the stronger assumption:

AXIOM 5. A data type has an effectively given basis.

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

$$e \prec e', \quad e \not\ll e', \quad e \sqsubseteq e'$$

are true and which are false. This certainly is going to require that the set E is at most countably infinite, and probably that we have 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. To make the data type really physical we would need everything to be highly computable 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

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

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 iff there is an effectively given subsequence.

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

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

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

In other words, 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. 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 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 $0,1,2,\ldots,n,\ldots$ (these elements are pairwise incomparable under $\sqsubseteq$) plus the two elements $\perp$ and $\top$, respectively above and below all the others. (A picture would help.) In this case the whole lattice is its own basis. For R the elements are closed intervals $[\underline{x}, \bar{x}]$ of ordinary real numbers ($\underline{x} \leq \bar{x}$) plus two elements $\perp$ and $\top$. The partial ordering between the intervals is defined thus:

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

The "perfect" reals are the one-point intervals $[\underline{x},\bar{x}]$ with $\underline{x} = \bar{x}$. The "approximate" reals have $\underline{x} < \bar{x}$. 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.

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

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

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

$$(x,x') \sqsubseteq (y,y') \text{ iff } x \sqsubseteq y \text{ and } x' \sqsubseteq' y'.$$

The sum is defined as a "disjoint" union of D and D', except that we identify $\perp = \perp'$ and adjoin a new $\top$" above all the other elements. (Again, a sketch of a lattice diagram will help.) The function space $(D \longrightarrow D')$ 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 D partially ordered in the obvious co-ordinate-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 given $a \in D$, the expression (a,x) defines a continuous function of D^∞ into itself. Consider a fixed point:

$$x = (a,x).$$

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:

$$x = (a, (a, (a, \ldots))))).$$

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 are not yet clear. A second example of the idea concerns function spaces. Let D be given, and set $D_0 = D$ and

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

The spaces D_n are (a selection of) the "higher-type" spaces 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 give D and is such that

$$D_\infty = (D_\infty \longrightarrow D_\infty)$$

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, 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 Curry-Church.

The reader should take notice of the fact that our abstractly presented theory of computable elements of lattices with effectively given bases applies to these 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 prgramming language. The λ-calculus itself is a programming language: it is the pure language of "unrestricted" procedures. It is only one of many possible languages.

In conclusion we can sketch the solution to the "storage-of-commands" problem mentioned in the beginning of this paper. 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 Σ of states of the store is defined by:

$$\Sigma = (L \longrightarrow V)$$

The space Γ of commands is defined by:

$$\Gamma = (\Sigma \longrightarrow \Sigma)$$

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

$$P = (V \longrightarrow (\Sigma \longrightarrow V \times \Sigma)),$$

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 take care of such an array of possible types of values we would need a fairly involved programming language.) We are thus led to write:

$$V = N + R + L + V^* + \Gamma + P.$$

If one substitutes the definitions of Σ, Γ, 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 type that we have understood previously in the "operational" way. We should now begin to

try to understand these languages mathematically, since we have all the tools necessary to do so.

ACKNOWLEDGMENTS. The idea of using monotonic functions in recursion theory the author learned from Richard Platek (Ph.D. Thesis, Stanford, 1965). The notion of continuity comes from papers of Lacombe, Nerode, Kleene and Kreisel and was used by Platek, but the theory has not been very much developed. The thought that there would be connection with programming languages occurred to the author in the course of joint work on program schemata with J. W. de Bakker in Amsterdam during the Spring and Summer of 1969. The idea of abstract lattices with bases is new and generalizes considerably the special structures employed by the above authors. The plan of finding a mathematical semantics for programming languages has been pursued for some time by Christopher Strachey, and it is due mainly to his stimulation and encouragement during the fall term in Oxford, 1969, that the author was able to make a coherent theory out of these diverse ideas. Technical papers on the λ-calculus and on lattices with bases will be published soon along with joint work with Strachey on semantics for various languages.