Outline of a Mathematical Theory of Computation

by Dana Scott (1970)

1 Introduction

Scott is attempting to provide a way of underastanding higher-level computer programs. He, as did Church, makes a distinction between functions as sets of n-tuples and functions as computations. Where Church went with the latter, Scott argues that the former is sufficient for *understanding* the computer languages in question.

> *The point is that, mathematically speaking, functions are independent of their means of computation and hence are "simpler" than the explicitely generated, step-by-step evolved sequences of operations on representations.* [1] (p. 1)

Thus, Scott is attempting to provide a way of understanding *computation* that leaves out the "irrelevancies"(i.e., the arbitrary choices involved in dealing with functions according to their computational definitions) of operational semantics.

However, Scott does realize that the approach argued for above is simply an argument for an approach that accomodates *human* understanding of computation and that the operational approach must not be ignored because, as he points out, the machines that the programs of study run on are not capable of dealing with such an abstract level of understanding. That is, the computaional approach should not be abandoned because the machines that we build operate on that lower level.

> *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.* (p. 2)

Consider functions, Scott states that functions defined mathematically can be known to *be* computable without knowing *how* to do the computation. However, "knowledge" of a function not only includes the abstract mathematical definition, but also its operational definition. That is, we must not simply know that a function is computable but also how to compute them (given input).

The point of this paper, is to provide a theory that accounts for the aforementioned *abstractions*. He points out that the operational approach has been used for some time (compilers are evidence for such understanding).

> *[T]he essential point : unless there is a prior, generally accepted mathematical language at hand, who is to say whether a proposed implementation is correct?* [2] (p. 2)

[1] By 'representation', Scott means that which stands for the information being functionally studdied (i.e., data).

[2] Notice that compliers do not provide us with such definitions for a particular language because there can be many compilers for that language

He points out that those who write compilers should have clear understanding of the language they are writing the compiler for and Scott's point is to say that that understanding should be made explicit. Why? So that such abstract entities can be "manipulated in the familiar mathematical manner."

Personal remark: It seems as though Scott has backed off of his "essential point." That is, the question he raises in making that point seems to be a definite problem for the operational approach. But later in that paragraph, when he brings up the point about compiler writers providing the standard (or at least having some sort of understanding of the standard), he seems to back off the implication that his mathematical approach will provide the standard for determining whether a program is correct and it appears that he is simply willing to claim that his approach is just mathematically interesting. That is, his approach doesn't *set* the standard it only makes the standard (hidden inside the heads of copiler writers) "visible" and that the reason why we should adopt his approach is only because it is of mathematical interest. Perhaps not. . .perhaps it is only vague language that I'm nit-picking on. We'll have to see if this vagueness continues as we go.

2 The Problem of Self-Application

When thinking about programming languages in this way we must have a new way of thinking about *data types* and the *functions* (i.e., mappings) that are to be allowed from one to another.

Since, a function can be in itself an infinite object (ex. a mapping from integers to integers) we must introduce some idea of *approximation* to deal with the finite nature of the machines (just as we do with real numbers).

Also, there exist operationally defined functions for which there is no mathematical counterpart. In particular it is "not unknown" in programming languages to allow *unrestricted* procedures which can very well produce unrestricted procedures as values. I'm assuming that by 'unrestricted' he means a function that is *self-applicative*.

However, Scott points out that, to his knowledge, no mathematical theory has been able to maintain self-application and remain consistent. This is another goal of the theory that he is proposing here.

A related problem for a mathematical approach to programming language understanding concern questions of keeping track of *side effects* and of *storage of commands*. Mathematically speaking, what is a store? Well, it's a mapping, σ, from information (contents) to locations. That is, it is a function that assigns to each location, $l \in L$ its contents $\sigma(l) \in V$ (the set of values).

$$\sigma : L \rightarrow V$$

What is a side effect? Well, its a change in *state*, S. What is a command? Well, it's a request for a side effect. Thus, mathematically speaking, a command is the following function

$$\gamma : S \rightarrow S$$

Can a command be stored? Well, suppose σ is the current state of the store, and suppose $l \in L$ is a location at which the command is stored. Then $\sigma(l)$ is a command; that is

$$\sigma(l) : S \rightarrow S$$

Hence, $\sigma(l)(\sigma)$ is well defined. Or is it? Consider the following:

A store, σ, is a function from locations in the machine to values. Thinking of functions in set theoretic terms, σ is simply the set of pairs of locations and values (state of the store).

$$\{\langle L_1, V_1\rangle, \langle L_2, V_2\rangle, \ldots\} \qquad \text{(i.e., } \{\langle L_1, \sigma(L_1)\rangle, \langle L_2, \sigma(L_2)\rangle, \ldots\}$$

A command, γ, is a function from one state of the store to another

$$\gamma(\sigma) \rightarrow \sigma'$$

If σ is the current state of the store and l is the location where the comand, γ, is stored, then $\sigma(l) = \gamma$. Thus

$$\sigma(l)(\sigma) \rightarrow \sigma'$$

Is this not well defined and self-applicative?

3 Data Types and Mappings

What is a data type? Is it just a set? That is, are objects *of* the same type if they are both in the same set? According Scott, that is just too simple,

> *[T]he objects [of the same type] are structured and bear certain relations to one another, so the type is something more than a set.* (p. 6)

The data types Scott is referring to are structures that are primitive, general structures (as opposed to data structures) that have to do with a sense of *approximation*. Consider $x, y \in D$ be two elements of the same type, not only are they different but that one is a better approximation of the other. Say,

$$x \sqsubseteq y$$

to mean intuitively that y is *consistent with* x and is (possibly) *more accurate than* x. For short, x *approximates* y. Data types should always be provided with such a relationship.

So, what can we say about this relationship?

Well, we want to assume that it is *reflexive, transitive, and antisymmetric*, which seems to be an intuitive assumption because it seems clear that $x \sqsubseteq x$, if $x \sqsubseteq y$, $y \sqsubseteq z$, then $x \sqsubseteq z$, and if $x \sqsubseteq y$, then $y \not\sqsubseteq x$. Thus, we have

Axiom 1: A data type is a *partially ordered set*.

Why only partial? Well, for example for any to objects, x, y, of the data type *integer*, $x \not\sqsubseteq y$. That is, no integer is a better approximation of the data tupe integer that any other element of that data type (except $\perp$).

If we have two sets, D, D', with the partial orderings $\sqsubseteq, \sqsubseteq'$, respectively and $f : D \rightarrow D'$ is a mapping from one set to the other, then if $x, y \in D$ then

$$x \sqsubseteq y \ \rightarrow \ f(x) \sqsubseteq f(y)$$

Thus, we have

Axiom 2: Mappings between data types are *monotonic.* [3]

As of now, I'm having a difficult time getting this. So, let's consider this: Think of the objects in the data type providing *information* for some entity. Thus, $x \sqsubseteq y$ means that x and y are attempting to approximate the same entity (i.e., they're consistent) and that y provides for information about it.

Let's try the following extremely intuitive example: The entity we are trying to approximate is *bachelorhood* and let $x =$ 'maleness' and $y =$'maleness and unmarriedness'. Thus, $x \sqsubseteq y$. They're consistent and it seems as though y is a better approxiamation of bachelorhood.

Scott points out, that by looking at data types in this way we must account for *incomplete* entities, like x in my example (I suppose it could be argued that y is also incomplete. But let's assume that it's not.)

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. (p.8)

4 Completeness and Continuity

The theory based on Axioms 1 and 2 would be too abstract. Why? Well, not only is Scott trying to give an abstract mathematical theory of computation, but he also wants to build a theory that can be translated into a practical implementation. So. . .

Suppose we have the following infinite sequence

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

then we want to suppose that the sequence is tending towards a limit. Call the limit y

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

This limit is to be taken as the *least upper bound* (l.u.b.). We can think of the limit as the union or join of each of the approximations. Thus,

Axiom 3: A data type is a *complete lattice* under its partial ordering. [4]

They're complete because since every subset of a data type has a l.u.b. implies the existence of *greatest lower bounds* (g.l.b.)

[3] A monotonic function is an order preserving function.

[4] A lattice is complete is *any* of its subsets have both a *join* (a l.u.b.) and a *meet* (a g.l.b.)

So, let $x, y \in D$. It may be that they are either both approximations of the same "perfect" entity or not (if so, then they're inconsistent). The l.u.b. of $\{x, y\}$ is denoted by

$$x \sqcup y \in D$$

Notice that that subset's l.u.b. is inside D (thus D must be infinite). However, not only does every subset of D have a l.u.b., so does D (denoted by $\top \in D$ and called the "top" of the lattice). Again, it is to be thought of as the union (or join) of all approximations. It is not that which is being approximated, but rather an "over-determined" element. Thus,

$$x \sqcup y = \top$$

means intuitively that x and y are *inconsistent*, as opposed to incomparible (dentoed as $x \not\sqsubseteq y$ and $y \not\sqsubseteq x$).

The l.u.b. of the *empty* subset is an element $\bot \in D$ (called the "bottom" of the lattice). It is the most "under-determined" element of D.

The equation

$$x \sqcap y = \bot$$

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

Since we have the notion of limit permitted by data types, we must re-think our notion of function.

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. (p. 10)

Scott claims that we should be able to express this idea in terms of the notions that we have available.

The proper notion of limit is best expressed 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 upperbound $y \in X$ so that we have

$$x_0 \sqcup x_1 \sqcup \ldots \sqcup x_{n-1} \sqsubseteq y$$ [5]

The limit of the directed set is the l.u.b., $\bigsqcup X$, and suppose that we want a finite amount of information about it. Each "bit" of information that we would need is contained in some element of X. Since, X is a directed set, all of the information about X is in at least one element of X. (Ya, but that element is the join of infinitely many things, right?)

[5] Notice that a directed set is alway non-empty

Now, consider a function, $f : D \rightarrow D'$ and given a limit $\bigsqcup X$ of a directed subset $X \subseteq D$, we ask for a "finite" amount of information about $f(\bigsqcup X)$. As was just intuitively shown, this would require a "finite" amount of information about $\bigsqcup X$ (i.e., a single element, $x \in X$). Thus, $f(x)$ should give us what we want

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

So, we could say that f *preserves* the limit and a mapping that preserves all the limits is called *continuous.*

Axiom 4: Mappings between data types are continuous.

Question: Why does Scott put quotes around the word 'finite' all of the time?

5 Computability

Again, we are at too high of a level, although some properties of computable functions have been isolated.

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."
(p.12)

GOOD! This is what I was wondering about in the last section. Let's see what he has to say.

Scott's solution to this problem is to take a topological approach. Any data type 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.* So, let D be a data type. A subset $U \subseteq D$ is open if the following two conditions are met:

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

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

Notice that $f : D \rightarrow D'$ is continuous in the limit preserving sense iff it is continuous in the topological sense. (Why?)

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

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

There are certain data types where certain elements are *isolated*. That is, where $x \prec x$ is true (ex. $\bot \prec \bot$ and $\top \prec \top$).

We write

$$x \preccurlyeq y$$

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

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

are successively weaker. (What are the differences?!)

We consider the possibility of having a *dense* subset of a space in terms of which all other elements can be found as limits, called a *basis*. A subset $E \subseteq D$ is a basis if it satisfies the following two conditions

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

(E_2) for all $x \in D$ we have $x = \bigsqcup\{e \in E : e \prec x\}$ [6]

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

Axiom 5: A data type has an *effectively given* basis.

That is, E must be "known." Given $e, e' \in E$, we have to know how to find the element $e \sqcup e' \in E$. We have to know how to decide which of the following are true and/or false

$$e \prec e', e \preccurlyeq e' \text{ and } e \sqsubseteq e'$$

This would require that E be, at least, countably infinite and effectively enumerated in terms of which the operations and relationships from above are recursive. (Is there a bit of circularity here? E must be physically implementable before we can *determine* it to be physically implementable. Maybe it's not circular, only trivia.)

The most important consequence of Axiom 5 is the possibility of defining what it means for for an element, of a data type, 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 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

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

[6] If a basis exists, then the *meet* operation ($x \sqcap u$) is continuous, and not necessarily otherwise.

This means that we must be able to give a better and better approximations to x which converge to x in the limit. This is essential, because the data type may have uncountably many elements, while there can only be countably many computable elements.

6 Construction of Data Types

We must consider now the construction of *useful* data types satisfying the axioms, remembering that the lattice structure is the most primitive structure on data types.

Notice that all finite lattices satisfy the axioms and are sufficient for practical application. However, many concepts are understood with infinite structures. Consider the data type, $\mathbb{N}$, for the integers. The elements of the lattice are $0, 1, 2, 3, \ldots, n, \ldots$ and $\top$ and $\bot$. Notice that elements, other than $\top$ and $\bot$, are *incomparable* under $\sqsubseteq$. That is, none of the elements "provide more information" than any other element. However,

$$\forall n \in \mathbb{N}(\bot \sqsubseteq n)$$

In this case, the entire lattice is its own basis. Why?

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

$$(D \times D'), (D + D'), \text{ and } (D \rightarrow D')$$

for obtaining new, "structured" data types from given ones. Let $x \in D$ and $x' \in D'$

$$(D \times D'): \quad \langle x, x' \rangle \sqsubseteq \langle y, y' \rangle \text{ iff } x \sqsubseteq y \text{ and } x' \sqsubseteq y'$$

$(D + D')$ is defined as a "disjoint" union of D and D', except that $\bot = \bot'$ and $\top = \top'$

The *function space* $(D \rightarrow D')$ has as elements all the continuous mappings from D *into* D', for which we define

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

Sums and products can be 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 ?sets? in the obvious coordinate-wise fashion. We can then set

$$D^\star = D^0 + D^1 + D^2 + \ldots + D^n + \ldots$$

which represents the data type of all finite *lists* of elements of the given D. Then we could do lists of lists of lists, . . . Now it would seem reasonable that D^∞ can be defined such that

$$D^\infty = D + (D^\infty)^\star$$

that is to say, that each element of D^∞ is either an element of D or is a list of other elements of D^∞.

Remember that every continuous (even monotonic) function mapping a complete lattice into itself has a *fixed point*. Thus, applying this to D^∞, for a 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 list x itself! Thus x is a kind of infinite list

$$x = \langle a, \langle a, \langle a, \ldots\rangle\rangle\rangle$$ [7]

How does this not fit our usual ideas about data types of lists?

One could say that D^∞ gives us the topological *completion* of the space of finite lists (How?), and the various limit points need not be used if one does not care to take advantage of them (Why?)

A second example of the completion idea concerns *function spaces*. Let D be given, and set $D_0 = D$ and

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

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 embedding each D_n successively into the next 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

$$D_\infty = (D_\infty \ \rightarrow \ D_\infty)$$

This space provides a 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 by an element of D_∞.

According to Scott, this is the first known, "mathematically" defined model of λ-calculus. How?

[7] Consider figure 4 on page 19

7 Conclusion

We can now provide an answer to the "storage-of-commands" problem mentioned above. Let L be the *location* space (finite or infinite). The space V of *values* is to be constructed by the limiting methods from above. Supposing it to already be constructed, the space, S, of *states of the store* is defined by

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

The space C of *commands* is defined by

$$C = (S \ \rightarrow \ S)$$

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

$$P = (V \ \rightarrow \ (S \ \rightarrow \ V \times S))$$

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.

What can those values be? Well, they might be *numbers, locations, lists, commands*, or *procedures.*

Thus,

$V = N + R + L + V^\star + C + P$ (i.e., values = integers + reals + locations + lists + commands + procedures)

Such a space V does exist mathematically, and it provides the values for expressions of a programming language of the kind we have understood only previously in the "operational way." We should at once begin trying to understand these languages mathematically, because we now have the tools to do so. (p. 21)