WHAT IS COMPUTATIONAL KNOWLEDGE AND HOW DO WE ACQUIRE IT?

D. E. STEVENSON

ABSTRACT. The goal of foundational thinking in computer science is to understand the methods and practices of working programmers; we might even be able to improve upon those practices. The investigation outlined here applies the methods of constructive mathematics *á là* A. N. Kolmogoroff, L. E. J. Brouwer and Errett Bishop to contemporary computer science. The major approach is to use Kolmogoroff's interpretation of the predicate calculus.

This investigation includes an attempt to merge contemporary thoughts on computability and computing semantics with the language of mental constructions proposed by Brouwer. This necessarily forces us to ask about the psychology of language. I present a definition of algorithms that links language, constructive mathematics, and logic. Using the concept of an abstract family of algorithms (Hennie) and principles of constructivity, a definition of problem solving. The constructive requirements for an algorithm are developed and presented.

Given this framework, the questions asked in the title may finally be answered in the conclusion.

1. Background

1.1. **Observations over the Years.** The ideas presented here are the result of observations over the years. Here are a few explicit examples.

- Requests of the form "I need a code in ⟨Language y⟩ to preform the same task as one already tried and true routine in ⟨Language x⟩." The latest incarnation was $y = C++$ and $x = Fortran$. Requests of this type completely negate libraries like `netlib`. I thought the idea was to solve it once and forget it.
- Recently, I needed a filled ellipse algorithm to a non-graphical use in a simulation. I thought I would be able to find an implementation that could be pirated. No. In fact, I could not find one in a textbook that was completely worked out. It was as if the problem had never been solved.
- While many computer scientists do not care, many do care about scientific programming. They seem to come in two kinds: those who know numerical analysis and not numerical methods; or the opposite. They are not the same subject, they do not use the same logical methods, and they do not have the same results.
- Somehow, computer science does not seem to get better in the same way that mathematics and the sciences do. In the sciences, experiments lead to new insights. In mathematics, new proofs lead to new insights. Computer science seems to be caught in the technology trap: do not rethink, just reprogram (see above).
- Programming in practice is not a series of unrelated exercises. One must model (in the scientific sense) first. Software engineering is finally seeing that their role in life is to produce models that can be programmed.
- Some issues must be looked squarely in the eye. The fundamental nature of computation is that it can fail. This is even true in classical mathematics: $1/0 = \infty$ because the uniqueness of the inverse of multiplication fails to hold. See Suppes' venerable book, *Introduction to Logic*[24]. ∞ is not a crisis — it is a reality to be dealt with.

These are not pie-in-the-sky problems, these are everyday, down-to-earth problems that affect on how our discipline is practiced. The question is, should theories about computer science worry about such things? Is it important that theoreticians try to say something about practice?

1.2. **What is the Role of Foundations.** It is often said that "philosophers of subject x cannot *do* subject x." That is, philosophers of science do not do science, they do philosophy of science. This is probably true, but philosophers *within* science x undoubtedly do science x. For example, Stephen J. Gould[12] certainly is a true biologist and he even does biology — but he also criticizes the conduct of biological research.

Mathematics has certainly had its share of philosophers over the past 200 years: Kant, Abel, Cauchy, Bolzano, Dedekind, Gauss, Kronecker, Weierstrass, Hilbert, Brouwer, Russell, and Poincaré to name a few. Computer science has its philosophers: Church, Turing, Kleene, Gries, Dijkstra, Denning, Scott to name but a few. The role such people play is four-fold:

1. They bring hidden problems to the surface for scrutiny.
2. They criticize and improve practice.
3. They formalize those things for which insight can be gained by such formalization.
4. They do not interfere with the free development of methods.

I certainly cannot put myself in the league of any of the above mentioned persons. But I can look at my discipline and criticize it constructively (no pun intended).

Since 1989, I have concentrated on the issue of *computational science and engineering*, defined as the study of the computational basis of science and engineering. Such a basis would have to be constructive and not classical[22]. This is not a particularly popular stand: the formulation of the λ-calculus is classical[1]. There are many arguments against a constructionist view, not the least of which is it is harder to work under constructive regimes. However, for the most part, mathematics is constructive: any proof that does not use excluded middles, non-constructive existence, completed infinite sets, or non-constructive omniscience is constructive (with a little work). But where does one come in contact with constructive theories outside analysis? A few examples exist like *NuPrl*. But there is little literature available at the introductory level to teach (indoctrinate) from. But where did we learn to build theories and prove things in the first place? Why, from *Euclid's Elements*.

I thus undertook a two year study of Euclid to determine if one could be a wild-eyed constructionist and live within the framework Euclid presented. And if that constructionist framework holds, what does it say about programming practices? The work here is an outgrowth of the Euclidean work[23]. The *Elements* is generally presented as the seminal text in axiomatic thinking: not so. The following observations/conclusions from [23] are important here:

- Most likely, the ancients were neutral on philosophical systems and their impact on mathematics.
- Constructive techniques are natural to the *Elements*.
- Axiomatic thinking is not a part of the *Elements*.
- The *Elements* is a teaching text.
- The *Elements* is not meant to be encyclopedic.

In the process of providing a foundation for the constructive basis of the *Elements*, I attempted to use 20th Century computing practices. In this way, I came to know how we might better understand programming from a constructionist point of view.

This paper is the first such analysis. My purpose is to look at the state of programming in the 1990s and shift through competing concepts to present a philosophically unified view. It represents a constructionist view of programming and program development.

1.3. **Introductory Musings.** This work concerns *problem solving* in its most general notion as it applies to computer science. Firstly, what does it mean to *be* a problem? Once I understand what it means to be a problem, can I then what it means to *solve* a problem?

The short answer is that this was all settled by Turing[26] in the 1930s. Turing's work was brilliantly extended by Kleene[15] so that by the 1950s most of the upper level concepts were solved or at least understood. Another flurry of activity followed Chomsky[8]. And let's not forget Petri. But the long answer is something much different.

The long answer is that, while there are highly theoretical results in many areas, the *methods* of describing problems and solutions are generally not available to the working computer scientist. I can draw a parallel to mathematics. The situation in computer science is analogous to a situation in physics before the calculus was known and physics models are constructed in an *ad hoc* manner. The impact of calculus on physics was deep and immediate. The rich development of analysis was fostered by its descriptive power of all manner of engineering and theoretical problems. Now, it is hard to argue that the *methods* of Church, Turing, Kleene, and Scott have had the same impact on day-to-day computing.

While complexity studies have had an impact on practical and theoretical concerns, it has not had the same impact as calculus in the sense that it has not given the typical practitioner methods of analysis and synthesis that the typical undergraduate engineer has. Part of the reason that computability studies have not lent more to practical problems is that computability answers a question in the philosophy of mathematics not computing: namely, Hilbert's Tenth Problem. The problem of characterizing the solutions of Diophantine equations is not on everyone's (anyone's?) list of what is holding back computer science.

> **My position is that the study of computability in the Church-Turing-Kleene-Chomsky-Petri-... sense is of virtually no help to working computer scientists.**

One purpose is to ask, "What do I really know about the process of development of computable (whatever that might turn out to mean) solutions to problems?" Our approach distills to one simple question: "What does it mean to say that program p solves problem P?" which I write as $p \vdash P$.

This is not a unique approach to the author—I took it from Kolmogoroff[16]. Even though Markov and Kolmogoroff were important early contributors to what might be called "constructive science", they had radically different ideas and approaches to computation. While the Markov model competed for attention with other views of computation, it was passed up for other abstractions. Kolmogoroff's impact was more in logic. I want to use his insights in a different way.

But there is more to the problem than $p \vdash P$: there are many cognitive aspects. While one might think of this as "soft," I claim that the process of *solution discovery* is inherently psychological. Programs and proofs only come after the mechanisms have been worked out *in ones head*. My second purpose, then, is to understand how the solution comes to be passed on from the originator to the general community. I do not take up question of "problem solving" in the psychological sense; *i. e.*, I do not ask how the psychological processes work but only how I communicate the solution.

2. Initial Concepts

Before going any farther, I present my concept of computation at this very high level. Computation is not defined in terms of processes (such as Turing machines) but rather is any discrete process with the following properties: Computation requires

1. a discrete, finitely presented[1] *state space*.
2. a discrete, finite, but not necessarily determinate, sequence of discrete, finitely presented *transition operations*.

I begin by considering the terms "data structures" and "algorithms." Now then, what does it mean to be a *data structure* or *algorithm*? That is, what problems do they solve?

- A data structure solves a *state representation problem*.
- An algorithm solves a *state transition problem*.

Thus, I need to know both state (data structure) and transition (algorithm). Let

$$S = (\textit{state space}, \textit{transition system}) = (St, Tr)$$

be the set of all systems that *solve* the original problem. This, of course, is not a new formulation[28].

How do I *know* that $S = (St, Tr)$ solves a problem P, Let $p \in S$; then we want to know if $p \vdash P$?

This is a question of (1) the problem space and (2) translation of the problem space to the encoded solution space. Let X be the input space for P and Y be the output space. Let $B(X, Y)$ be all the *behaviors* that satisfy the problem. A behavior is some mechanism that solves P, not necessarily computable (voodoo, for example). If we assume X and Y are not countable, then we need input and output spaces that are computable. Let $\hat{X}$ and $\hat{Y}$ be those spaces and let there be two *encoding* functions $\tau : X \to \hat{X}$ and $\tau' : \hat{Y} \to Y$. Finally, let $b : B$. We define $s \vdash P$ by the below diagram

$$\begin{tikzcd} X \arrow[r, "b"] \arrow[d, "\tau"] & Y \\ \hat{X} \arrow[r, "p"] & \hat{Y} \arrow[u, "\tau'"] \end{tikzcd} \tag{1}$$

That is, if $B = \tau' \circ s \circ \tau$. Notice that Eq 1 is slightly different than the usual category-theoretic formulation which would have

$$\tau' \circ B = p \circ \tau.$$

This formulation emphasizes one of the problems in numerical processing: the interpretation of computer output as if it were done in the infinite systems.

There are only two questions about a solution:

1. How to prove that p really does solve P?
2. What physical resources are required?

There is really only one technique for finding solutions: that of *continuous improvement*. In continuous improvement, you find a inelegant, inefficient partial solution p and then use correctness preserving transformations and complexity analysis to build a more efficient solution p'. Heuristically, then, I see that the solution p comes from understanding the *shape* of X and Y, the encoding/ decoding pair (τ, τ'), and the behaviors B. Therefore, we define *solves* in term of Diagram 1.

2.1. **Proof.** Diagram 1 defines the meaning of $p \vdash P$. Using the language of models (science), we would say that $p \vdash P$ if we can validate p against the problem P, where validate means to measure up in every way (see Section 5.1). Since computer science assumes the stance of being a deductive (not inductive) science, then I must define what I mean by *truth* in this system. Let $b : B$ be a behavior.

1. $p \vdash P$ if there is no $x : X$ such that $bx \neq \tau' \circ p \circ \tau(x)$. This is the "no counterexample" (Herbrand) definition. This would also be the testing viewpoint.

[1] This means states are recognizable by potentially real and processes. The question of finite complete presentation is taken up later in the paper.

2. $p \vdash P$ if there is a proof theory so that p is derivable in that theory. This is the program proof viewpoint.

In general, I may not have any complete characterization of B, the set of all problem solving behaviors. I must assume I have at least the ability to discriminate, on a case-by-case basis, whether or not the problem is satisfied for a given input and output.

2.2. **Performance.** It would be easy to dismiss performance as outside the scope of the investigation; indeed, space limits a close inspection. It is clear that there are "only" two criteria: minimum time (sequence and transition time) and state space representation. These are really engineering criteria.

Before dismissing the subject entirely, though, I want to make an observation. In practice, programmers often focus on performance and do not ask the question, "Is it right?" nearly often enough.

2.3. **Linking to the Literature.**

2.3.1. *Notation.* I have a reasonable vocabulary on computable solutions. A *procedure* is some entity (to be dissected later) that takes inputs, eventually halts, and produces an answer. I denote the statement *p halts* as $p\downarrow$ after Hennie[14] and *p does not halt* as $p\uparrow$. In equational form I write $pa = b$ and $pa = \perp$ respectively, where a is the input. For relations, I specify that the range is over the boolean set $\{0, 1\}$.

Totality and partiality are as usual. Numerical programming reminds us there are two ways to be partial: there is the nonterminating version and the terminating, but incorrect answer version. The latter happens all too frequently in real codes and is very insidious.

2.3.2. *Problems.* "P is a problem" is a judgement. The judgement here is that there is at least one computable solution to P. In other words, P is computable. Then there is a procedure p such that $p \vdash P$.

It turns out there are some sets worth mentioning here.

dom P. All the possible inputs formable from P's data types.
ran P. All the correct answers for P.
coran P. The range over which $p \vdash P$ works.
codom P. All possible outputs formable from P's data types.
$\mathtt{S}$. States.
$\mathtt{B}$. Behavior.
$\mathcal{D}$. Data types and their canonical values.

The statement P represents a problem if one can *decide* (in a definitional way) on the domain, co-domain, co-range, and range such that in principle one can make the set definition of $\mathtt{B} = \{\langle a, b\rangle\}$ where $a : corange$ and $b : range$. $\mathtt{B}$ is a problem if I can translate $\mathtt{B}$ into the *abstract family of algorithms* $\mathcal{AFA}$[14] as shown in Eq 1. I want constructive solutions that would give behaviors $h : B$ so that $h\ a\ b = true$. If we think of the problem in a model-theoretic sense, then we would say that the pair $\langle a, b\rangle$ satisfies the problem P or $h\ a\ b : Y$ so $b \vdash P$. Under the state-transition concept, the computation proceeds by state changes which should ultimately lead us to topological concepts.

3. Looking into the Psychological

Having established a link to computing vocabulary, it is time to consider the psychological.

3.1. **What is an Algorithm?** The question of what is an algorithm has been avoided for some time. It is nice to leave it open — I do not want to over-restrict ourselves on the form. Several years ago, I decided to look at definitions of *algorithm* that appear in randomly chosen, published texts. Every text has its own version; I would be remiss if I did not add to the collection:

> An *algorithm* is a finite sequence of effectively computable operations acting on finite constructions that solve a problem.

The problem, noted in [22], is that not being able to define what an algorithm is makes us unique in the mathematical sciences.

What should I consider in taking a stand on a definition? Here is a short list: I

- Must have the meaning of each term.
- Must be able to prove termination.
- Must be able to judge partial and total correctness.
- Must obey the Church-Turing-Chomsky property.
- Will be either a partial or total relation.
- Must be able to transmit algorithms to others.

This last is a psychological issue and generally taken as outside "mathematical concerns." I disagree — this is indeed the crux of the matter.

There is a certain formalness required in well-defined algorithms. But the real questions are

1. What does it mean for someone to *know* an algorithm?
2. What does it mean for someone to be able to *apply* an algorithm?

In classical mathematics, knowledge is taken as synonymous with provability. That is, one knows something only when one can demonstrate a proof of the statement. But this begs the question since one can often demonstrate a proof without any understanding of what the proof says — and hence not *know* in the usual sense. This particular problem is what helped lead Hilbert to formulate his 23 problems. Hilbert wished everything to hinge on provability and deterministic rule application. Hilbert's protagonist, L. E. J. Brouwer, no mathematical slouch himself, challenged Hilbert's view. Brouwer said that *personal mental constructions* of mathematical objects was the foundation of mathematics. Brouwer left hanging the question of sharing those constructions.

The second question, application, brings up the problem-solution relation. To apply an algorithm, I must first decide that the algorithm actually does something useful in solving the problem at hand.

My answer to these problem lies in following Brouwer — taking the constructive route.

Definition 3.1. *An* algorithm *is a pair (effective operation sequence, constructive proof).*

This definition is only satisfying if I can answer the following questions:

1. How do I know what I know?
2. How do I know that you know?
3. How do I transmit what I know to you so you know?
4. How do you transmit to me so I know I know?

Taking this view leads to an answer to the question of "What does an algorithm mean?" I address this issue below.

3.2. **Understanding Algorithms Psychologically.** To understand the knowing and transmitting of algorithms, consider what happens when a student enters a classroom for the first time. In the U. S., many of these students will have extensive backgrounds in computing. But I want to think of the student who is quite bright but has no background in computing. What is so hard about getting such students up to speed, as it were? Primarily, the problem is that such students do not share with the instructor any sort of mental images. For example, the student probably has never computed the sine of the angle $37°27'$ by means of either a series or a sequence of half- and

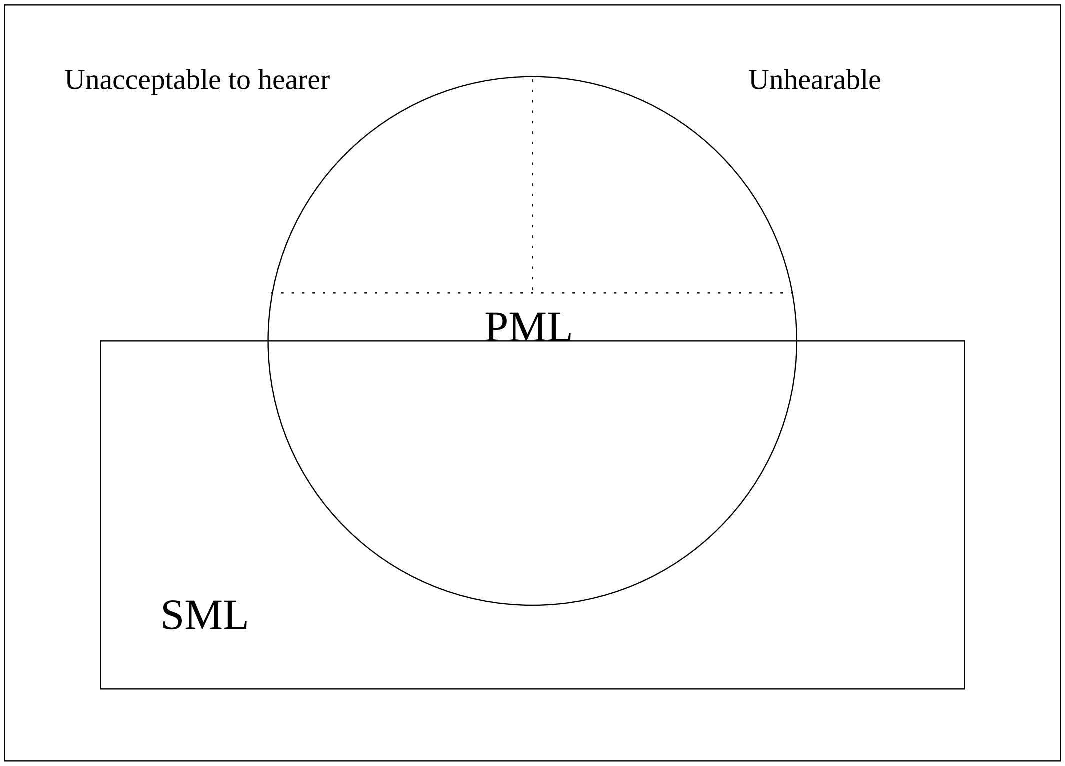

FIGURE 1. What Goes On

double-angle formulas. The student has not done much of anything algorithmically: sorting integers alphabetically is something that does not get much emphasis in high schools. To communicate with such students I must build shared vocabulary. How does such a vocabulary get developed?

In the instructor's mental *constructions* there are images that I shall call *personal algorithmic constructions* (*Pac*). Consider how one might think of the computation of a simple recursive function like factorial. Until one can visualize how the bookkeeping aspects go, the usual definition is difficult to comprehend:

$$\begin{aligned} 0! &= 1 \\ n! &= n \times (n-1)!. \end{aligned}$$

To transmit these facts to the students, I must have a set of *shared algorithmic constructions.* Until the instructor and the student share these images, no understanding can take place. We can also have mental images that are not algorithmic; these would be *personal mental constructions* (*Pmc*) and *shared mental constructions* (*Smc*), respectively. I can attach names to these constructions and hence, language is born in the subject matter. It is the *language* and its associated objects that must be learned.

The shared constructions are built by three basic means[27]:

1. Active, ostensive acts;
2. Passive, verbal/nominal association; and
3. Feedback using information content.

Active, ostensive acts are "show me" acts. These can be reasonably subtle. Passive acts are the method most used in mathematics: a nominal definition attaches a name to a formula that may or may not be meaningful without further development. I use information content here in a technical sense: information is the ability to discriminate between sets.

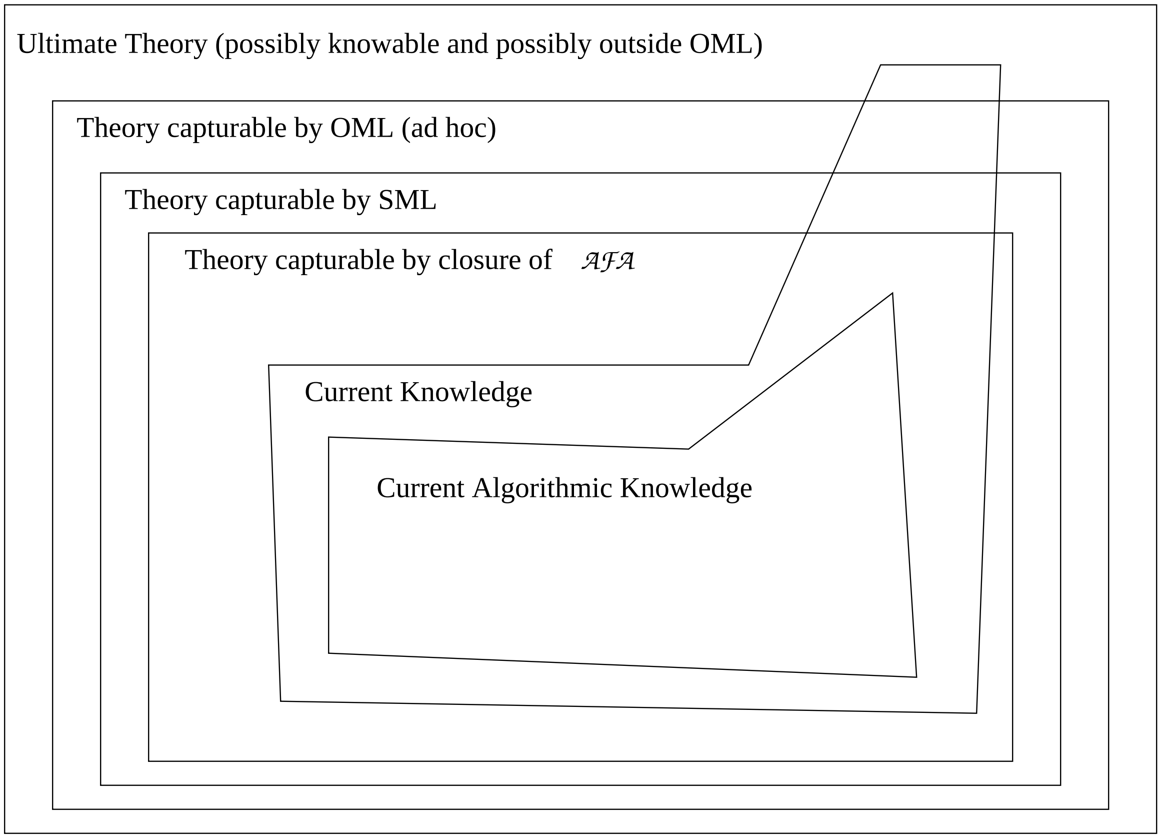

FIGURE 2. The Language Hierarchy

To acquire computational knowledge, then, requires the following:

- Personal algorithmic constructions (*Pac*).
- Shared algorithmic constructions (*Sac*).
- Lexical/linguistic associations attached to constructions.
- Active, ostensive acts
- Passive, verbal associations
- Feedback
- Information

These ideas help generate what are called *models* or *paradigms* in science. Surely, language of science is the language of models and paradigms[17]. The sum total of all these considerations lead to the issue of language.

3.3. **Language.** There are all types of natural language based on intended use, including poetic, theologic, romantic, ... I call this *open language* or *natural language*. Open language is loosely defined and admits to all sorts of misunderstandings.

Pac and especially *Sac* are not well captured by open language. This problem has been at the heart of logic from before Aristotle. Hilbert called the language of a subject a *metalanguage*. This is decidedly not Milner's `ML`. I most often see critical attempts to separate open language with metalanguage in logic. I define four levels of metalanguage (Figure 2):

1. The open metalanguage (*OML*) is the informal language that captures all aspects of the subject. Everyone has her/his own OML. It is informal and is used to discuss speculations, *etc.* This leads to the requirement there be a shared metalanguage (*SML*).

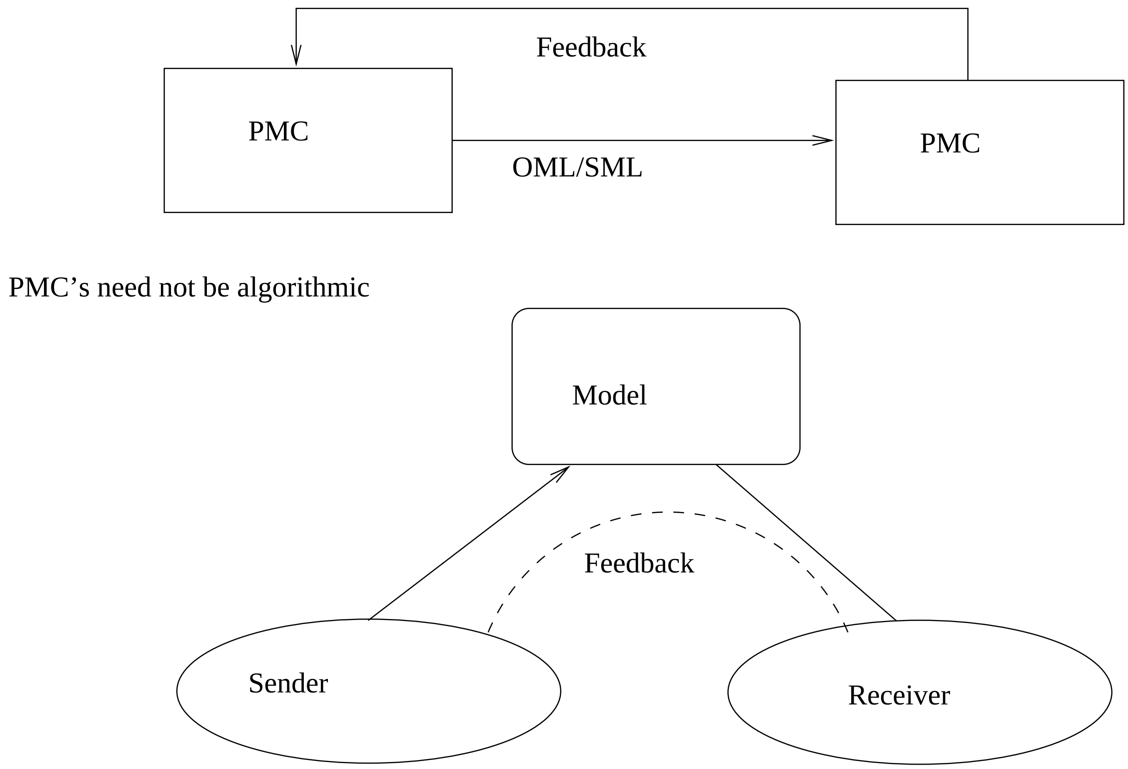

FIGURE 3. The Development Feedback Loop

2. The closed metalanguage (*CML*) is the formal language associated with the subject. *CML* encapsulates all formally recognized aspects. *CML* must encompass *Sac*. There is also a personal metalanguage (*PML*); otherwise, progress is not possible.
3. The intersection of all this must include the algorithmic family of languages ($\mathcal{AFA}$) [14]. The λ-calculus is technically an $\mathcal{AFA}$, but it is often used as an *OML*.

It is easy to see why the student is confused: all these metalanguages are used, more or less simultaneously, in class.

The personal and shared metalanguage interact (Fig 2). Progress is made by extending the *PML* into the *SML*. The development of new modes of solution are developed through feedback and information (Fig 1 and Fig 3).

My offered conclusion is that models are what we strive for and we seek to have a common, shared metalanguage that is as inclusive as possible.

4. ALGORITHMS

It is clearly the intention that $\mathcal{AFA}$s serve as the language that contain algorithms related to the system at hand. I have pointed out already that this language is used quite informally at times. For example, the `lambda` used in Lisp has taken on several different informal uses — namely, `expr`s and `fexpr`s until the recent formalization adding `lambdaq` to the repertoire. My use of the term algorithm has two new connotations.

The first is that an algorithm must have a problem-solution connection. Whenever I consider an algorithm it is in respect to a problem that it solves. This is the Kolmogoroff connection. This is not much of an addition.

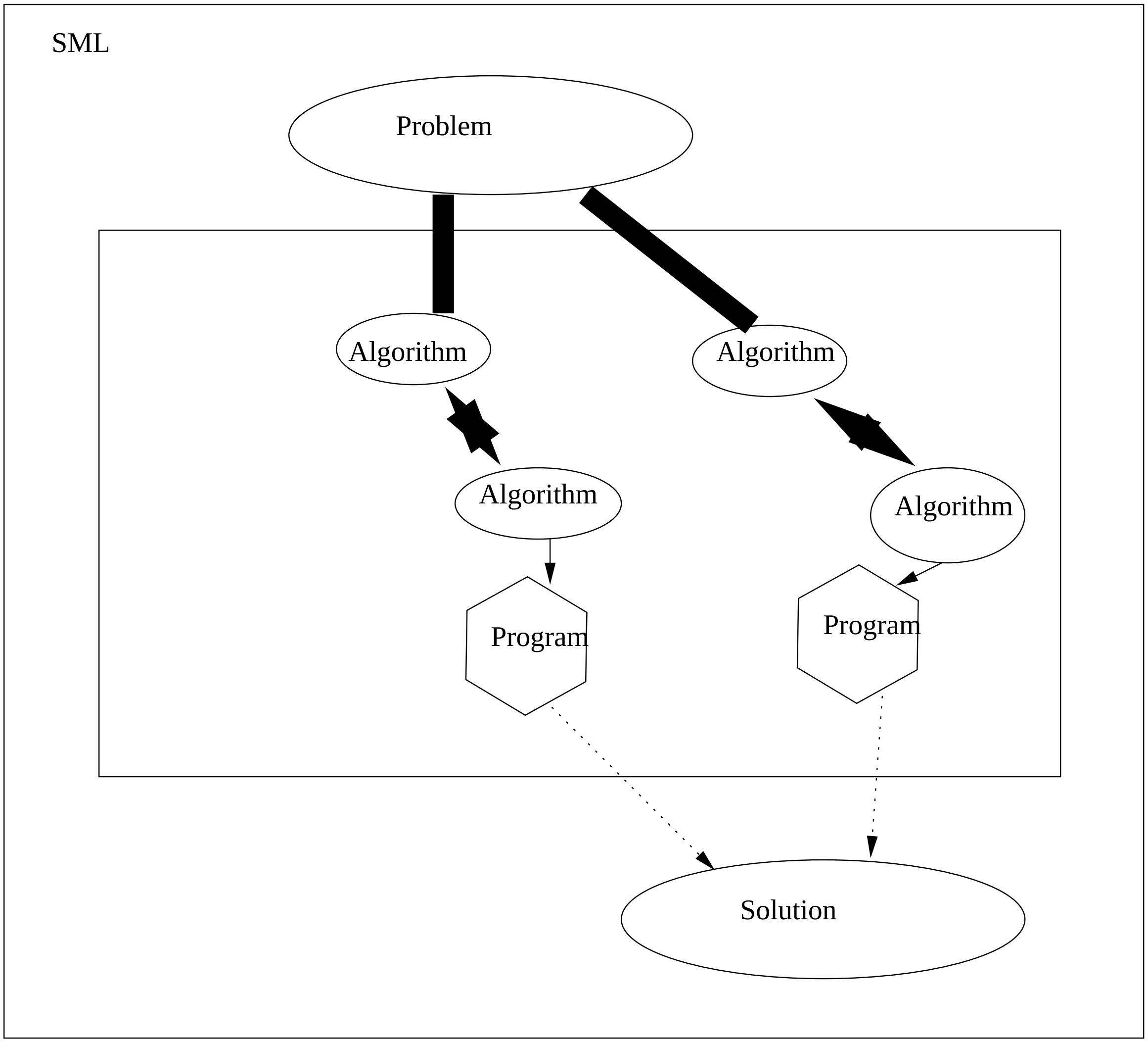

FIGURE 4. The Kolmogoroff View

The second connotation is really a subtraction: when is something that appears to be an algorithm no longer an algorithm? I take the view expressed by James Fetzer in "Program Verification: the very idea"[11]. Fetzer's argument, for the record, is summed up as

> The notion of program verification appears to trade upon an equivocation. Algorithms as *logical structures*[emphasis mine], are appropriate subjects for deductive verification. Programs, as causal models of those structures, are not. The success of program verification as a generally applicable and completely reliable method for guaranteeing program performance is not even a theoretical possibility.

An algorithm is no longer an algorithm when it is being executed on a computer.

I call this the *Fetzer Boundary* (See Figure 5). Whenever the algorithm is executed by a physically realized machine, the guarantees of the formal world are lost. The computing agent is different in kind, not language.

Thus, the Fetzer boundary also represents the lower limit of abstraction. The agent used to execute the algorithm must have concrete representations to manipulate. Is there any point at which an algorithm is too abstract to serve as an algorithm? Using our definition 3.1, the upper limit is reached when the proof cannot be sustained constructively.

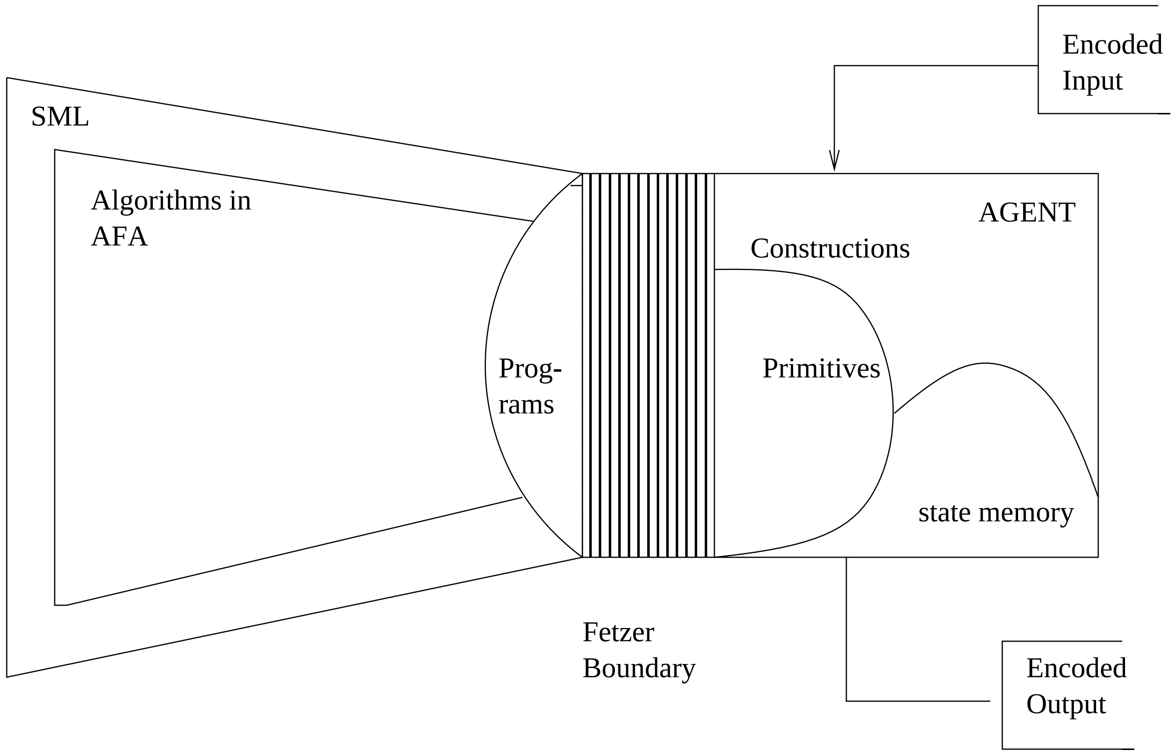

FIGURE 5. The Fetzer Boundary

The understanding of Fig 5 — algorithms — and the basic idea that the focus is on $p \vdash P$ can be seen in a bottom up fashion:

1. Computation by an agent. The computational agent follows instructions according to some partial order. Instructions either test or alter states. The fundamental concept for a single agent is the *and-then* operation. Therefore, the three fundamental ideas are (1) instruction, (2) state, and (3) order or ISO.
2. Programs. Programs are statements in a formal language that convey ISO information. A program p represents a proposition P that the problem P is fulfilled by the objects c constructed by the program p. That is, the program p is the directions for constructing c. c demonstrates that the properties of the proposition P can be computationally verified on any applicable object. Alternatively, I can think of P being a contract, p is the fulfillment manual, and c is the product.

   In either case, c gives meaning to the proposition P in the Martin-Löf sense. The assertion $c \vdash P$ should be seen as asserting "c fulfills P contract". The program p and the construction c are developed by constructionist rules. See Section 5.
3. Correctness. Suppose the algorithm in question is A. Let $fix\ A$ be the fixed point of A and $tr\ fix\ A$ be the trace. Then $tr\ fix\ A$ is the complete test for A, less the issue of encoding; *i. e.*, there could be a translation step for input and output values. Now, where do I need testing and where can I use proof? In Fig 5, proof can only be meaningful *to the left* of the Fetzer boundary and test is only meaningful on the right. A discussion of *test versus proof* is in section 2.1.
4. Algorithm. I have never officially defined *algorithm*. An algorithm is an object, so it must be amenable to manipulation. It should therefore be a statement in a formal language. But to obtain meaning, the algorithm must guarantee constructions. Therefore,

$B1$ A proof is any completely convincing argument[a].
$B2$ Proofs must have [computational][b] content.
$B3$ *Preset* requires procedure to construct arbitrary element.
$B4$ *Set* is a preset equipped with *equality*.
$B5$ No excluded middle[c] or impredications[d].
$B6$ No assumptions of omniscience[e].
$B7$ Construction precedes use.

[a]Bishop disliked formalized approaches such as Heyting's. Our concept of algorithm demands a formal language.
[b]change by me from "numerical".
[c]Excluded middle is a modeling problem. If objects possess certain properties, then exclusive or's are perfectly legitimate.
[d]Impredication is assuming an object exists without having constructed the object first.
[e]Omniscience is the assumption that the sets are completed and I know ahead of time a particular condition is true.

FIGURE 6. Bishop's Rules for Mathematics

Definition 4.1. *An* algorithm *A is a statement ϕ in a formal language ("the language of construction") and a proof ρ such that the object computed by ϕ fulfills the proposition of the algorithm $\mathcal{P}$ as witnessed by ρ.*

5. The Problem. Note that it is meaningless to talk about an algorithm without also talking about its associated problem(s). From elementary considerations $A \vdash P$ is not a canonical judgement.
6. Rules of the Language Game. Everything not explicitly stated in the algorithm (*i. e.*, either in ϕ and ρ) is in the metalanguage. The questions of *adequacy* and *meaningfulness* are discussed in Section 5. Thus, algorithms are explicit with respect to a metalevel. This includes the definition of the $\mathcal{AFA}$.

We have briefly touched on the concepts needed in a comprehensive theory of computing. In Section 5, I describe the received concepts of constructive mathematics. In Section 6, I describe the elements of constructive computing.

Since I have introduced language, I need to say what role it plays. The purpose of the formal language is to compress state-transition sequences into a finitely represented entity. In the introduction of any such language we trade explicitness for implicitness; the implicitness goes into the metalanguage. This has deep consequences in practical computer science.

5. Principles of Constructivity

5.1. **Received Principles.** To set forth a consistent set of principles for constructive programming, there are certain concepts that I inherit from other aspects of constructivity. A succinct list of rules gleaned from Bishop's writings[2, 3, 5, 6, 4, 7] are shown in Figure 6 and expanded below:

1. Objects. Objects in programming are figments of the imagination of humans but representations stored in a computer. The first view is from Brouwer and the second from the reality of computers.
2. Existence. An object cannot be used until it is constructed. For example, de-referencing a pointer variable before it has anything to point to is a "bus error".
3. Effectiveness. The concept of effectiveness is that of substitution and rewrite. Although computer programs might not work exactly that way, most humans do and certainly the received computability theory demands effectiveness.
4. Martin-Löf *Proposition as Types* Principle. The propositions as types principle is essentially the next step in constructivity after Bishop. The concept of proposition used in Section 4 is an implementation of the Principle. As part of the proposition as type principle is the

stated goal of removing as much from the metalanguage as possible; *i. e.*, positively stating all information.

5. Brouwer-Heyting-Kolmogoroff interpretation. I have already alluded to the Kolmogoroff version.
6. Failure. All constructive theories explicitly acknowledge that something might fail to exist.

Constructivity as a principle extends from a 20th Century philosophical principle called "verificationism". For the most part, I are uninterested in verificationism except in one major point: the focus on meaning. Feferman[10] develop two criteria for what theories *should* accomplish instead of the mechanics.

$F1$ A theory T is an *adequate formalization* of [a body of of informal mathematics] M if every concept, argument and result of M may be represented by a (basic or defined) concept, proof, and theorem, respectively, of T.

$F2$ A theory T is *In accordance with* or *faithful to* a [body of of informal mathematics] M if every basic concept of T corresponds to a basic concept of M and every axiom and rule of T corresponds to or is implicit in the assumptions and reasoning followed in M (*i.e.,* T does not go beyond M conceptionally or in principle).

The important point of these two criteria is that provability is not equivalent to meaningfulness nor is meaningfulness merely a synonym for practical.

6. Elements of Constructionistic Computing

The task now is the top-down development of the elements of constructive computing.

Late 20th Century views of constructionism/intuitionism are succinctly present in [1, 2] and the papers of Martin-Löf. A very brief overview of those principles is developed here.

A set is seen as a *container* that holds elements that fit a particular definition. That definition may not admit anything so that a constructive set must be shown to be *inhabited.* Elements are objects and the definition is a constructive procedure. The object usually has parameters which describe the object; these parameters are called *witness data.* It is the witness data that is the focus of constructive processes. When one explicitly accounts for witness data, we say the data is *fully presented.* Full presentation is a literary nightmare. However, that is exactly what the programmer is faced with.

With this in mind, the elements of constructive computing would describe how objects are formed and how the witness data guarantees that the constructed object is the desired object. There is an interesting tie-in with [23]. Diagrams have been portrayed by purists as unnecessary and should be eliminated. In [23] we give another interpretation: the diagrams are the blueprints of an object. The auxiliary lines, *etc.*, of these diagrams are the means by which we hang the proof off the construction. Similarly, rather than banish information from programs we should encourage the use of annotation. Programming languages need to admit annotation and encourage its use — without this, we will continue to have the sorry state of affairs regarding reliability, cost, and enhancement of software.

6.1. **Principles for Abstract Families of Algorithms.**

The actual structure of the $\mathcal{AFA}$'s have not been studied in computer science except for the various forms of the λ-calculus. Under my thesis, the $\mathcal{AFA}$ is the center of attention because it serves to capture the metalanguage. Clearly, the $\mathcal{AFA}$ must be sufficiently rich to capture Feferman's criteria. One might think of $\mathcal{AFA}$'s as the obvious basis of object-oriented systems. Where the OOP concept focuses on the pair *(data, method)* the $\mathcal{AFA}$ focuses on underlying semantic rules needed to describe *(data,method).* The $\mathcal{AFA}$ must capture the meta operations shown in Figure 7

Alphabet definition
Lexical rule specification
Syntax rule specification
Rule definition
Abstractor (binder) definition
Substitution and rewrite
Computational ordering
Proof statements
Annotation, Tacticals and proof manipulation
State manipulation
Canonical constant generation and singular terms

FIGURE 7. Common Meta-Operators in $\mathcal{AFA}$'s

6.2. **Algorithms.** Specific algorithms are represented in their own meta-system. The meta-system captures, among other things, the assumptions being made about the system. Therefore, we say "algorithms are explicit in their own meta-systems."

Algorithms are formal objects in their metalanguage. To implement the Kolmogoroff process depicted in Figure 4, I operate with two shift principles that are obvious generalizations of the β rules in the λ-calculus.

1. An *intension-extension* shift. This is a binding of specific values (extension) to specific variables that are abstractor-protected (intension). By implication, all intensional statements become "true" and hence are removed from consideration.
2. An *extension-intension* shift. This removes a value (extension) and replaces the extension by an intensional variable (and abstractor) and inserts any intensional rules.

These two operations must be explicit in the $\mathcal{AFA}$. To be constructive, it must be provable that $c \vdash P$.

The pragmatics of algorithm development is such that, by and large, proofs of $c \vdash P$ are difficult to develop. Various concepts have arisen to deal with this problem, the most common is a command often called `assert` which *dynamically* tests the condition for continued execution. This actually leads to a theoretical challenge I call the *static-dynamic* principle:

whatever cannot be proven statically must be asserted dynamically.

For the most part, this principle is a consequence of the two shift principles above.

What is *full presentation*? In this context, an algorithm is fully presented relative to its metalevel. This leads to establishing the status of such concepts as annotation, substitution and rewrite, tacticals, *etc.*

6.3. **Basic Principles.** Compilers are programs that implement the metalanguage and hence address the issues in Figure 7. Unfortunately, as we well know, the understanding is also buried in the compiler. There are several extant systems of specifying semantics; unfortunately, like the blind men describing the elephant, each tale of the element depends on the teller. This is a reality and not a negative criticism. But when do semantic systems impact practice?

I believe it is fair to say that most systems of notation that are likely to be used in programming languages for the foreseeable future are those that can be used to develop directed acyclic graphs of the statements. Even the most radical machines on the drawing board are effectively multiple von Neumann systems. Adopting the denotation paradigm of graph-evaluator-semantic algebra does not seem to present a constraint. So what? The current models of programming do not allow (force?) the programmer to write down intensions, conjectures, thoughts *etc.* along with the technical presentation. Yet we would never accept this paradigm in a mathematics or computer science textbook. This minimalist presentation prevents any form of growth in the understanding of the program and impedes development. One can only hope that we find more intellectually efficient methods. Why do we program in such an intellectually sparse manner? My guess is that history is to blame. Early compilers didn't have the time or space to deal with other issues. But this is the 21st Century, not the 1960s.

The usual model for programming is something based on the λ-calculus, going back to McCarty[18]. This model represents the "unwinding" version of the λ-calculus. Hence the meta-language rules have those operations, stacks, *etc.* The thesis presented here is that algorithms have both state and witness data manipulations. Therefore, we must define the metalanguage rules so that both are presented: (1) the usual state machine and (2) a modified proof in the spirit of the static-dynamic principle. Using LCF[19] and NuPrl[9] as exemplars, the system could be seen as a mapping

$$\langle A_i, \Sigma_i, W_i \rangle = \langle A', \Sigma', W' \rangle$$

where A are algorithms, Σ are states, and $W = \langle \textit{Proof, Hypotheses, Tactical} \rangle$.

Taking or cue from the previous section, the rules consist of static, compile-time operations describing the computational stack and the proof. The proof is carried out at compile time as best it can, realizing that failure is an option. the proof identifies hypotheses that must be asserted for the proof to complete. Failure of any hypothesis is failure for the program.

7. Semantics and the Case for Construction

There are several ways to present semantics. The three most common — as the reader is well aware — are (1) operational, (2) axiomatic, and (3) denotational. See, for example, Gunter[13]. It is now common to used category-theoretic notation to describe denotational theories as we did in Equation (1). To provide a constructive basis for semantics, however, we return to the original concept of *solves*. Recall that I started out with a discussion of Kolmogoroff's concept. This suffices for the semantics in its logical interpretation but not in its programming concepts. Rather than *solves* the key issue is *approximates.* This is, of course, Dana Scott's original insight.

7.1. **What About Meaning?** Meaning, of course, is a loaded word and leads to philosophical problems almost immediately. Such was the case at the turn of the Century leading to analytic philosophy and the Vienna Circle. The Vienna Circle was led Moritz Schlick[2], a "retred" physicist. Schlick's stand was that the sciences and mathematics were not meaningless symbols but had to be coordinated with reality. (This is different than Brouwer, but a conclusion in [23]). In computing we are faced with an equivalent problem because a program is a model of some system. What we compute must match what the real system does. Validation, in the sense of model validation, determines whether or not the program's "system" and the system presented initially are connected. A computational physicist cannot negotiate with Nature so that her model is easier, more efficient, or more elegant. Likewise, artificial intelligence systems, accounting systems, air traffic control systems, medical diagnostics, and fingerprint matching programs have a reality they have to validate to.

Meaning, therefore, is the match of the computation and the evolution of the system. The central view of meaning is the theme of Waismann's[3][27]: where the *meaning* of a construct is taken as its use. A more natural way to say this might be that *meaning* derives from use or effect. Considering Feferman[10] again, we can say

$M1$ An operation ϕ is an *adequate formalization* of an operator $\mathcal{F}$ if every feature of $\mathcal{F}$ is represented by ϕ.

[2] His leadership was cut short by his death in 1936; killed by a student over his Jewish heritage.

[3] Ranta[20] suggests Waismann as a key to understanding analytic/linguistic philosophy

$M2$ A operation ϕ is *faithful to* an operator $\mathcal{F}$ if every basic concept of ϕ corresponds to a basic concept of $\mathcal{F}$ and every rule of ϕ corresponds to or is implicit in the assumptions and reasoning followed in $\mathcal{F}$ (*i.e.,* ϕ does not go beyond $\mathcal{F}$ conceptionally or in principle).

While stated in terms of operators, the same concepts apply to relations.

8. Constructive Details

8.1. **Basic Concepts.** Our guiding concept is Diagram 1:

$$B = \tau' p \tau.$$

This means that the witness data come from the τ transition. Let $a : A$ be an *element* of the set A. Let $\hat{a} = \tau a$ be a completely presented *object* of the computational set related to A, which is denoted $\hat{A}$. We come back to dealing with sets after we intuitively understand elements. For $\hat{a}$ to be completely presented there must be information about its origins; I write this $\hat{a}\langle\alpha\rangle$.

There is an obvious assumption: the cardinality of A is greater than the cardinality of the set of constructible objects. Therefore, τ is not invertible and τ' is set-valued. Therefore, $\hat{a}$ stands for the set of domain points that map to it.

Now consider the identity function. This should give $id = \tau' \circ \tau$. I use this to define $\hat{a} \vdash a$.

Definition 8.1. *$\hat{a} \vdash a$ if (i) $\tau a = \hat{a}$, and (ii) $\hat{A}\tau a = true$, and (iii) $a \in \tau'\tau a$. $\hat{A}$ is the membership function for the set as explained in Section 8.2*

It is not clear what the status of τ and τ' are: they have a foot in each world. Taking a cue from analysis[21, p. 42], I call the set $\hat{a}$ a *cover.*

8.2. **Sets.** Since $\hat{A}$ is constructive, there is a function that passes on membership in $\hat{A}$. To reduce notation, assume that $\hat{A}$ is that function. In [25], the convention is that $\hat{A}\hat{a}$ is *true* when $\hat{a}$ is an object in $\hat{A}$ as in (ii) above. Actually, this convention seems to be handy about one-half the time; the other half $\hat{A}\tau a = \hat{a}$ would be better. Staying with the first convention, I take $\hat{A} \vdash A$ if for all $a : A$ $\hat{A}\tau a = true$.

In more familiar terms,

$$\frac{}{a}$$

introduces a as a canonical element and

$$\frac{}{\hat{a}}$$

introduces $\hat{a}$ as a canonical *object* with the implied witness of $\hat{a}\langle\alpha\rangle$. The equivalent judgements are

$$\frac{a : element \quad \in_A : membership \quad \in_A a = true}{a : A}$$

$$\frac{\hat{a} : object \quad \hat{A} : object \to boolean \quad \hat{A}\hat{a} = true}{a : \hat{A}}$$

So far, though, we have not said how $\hat{A}$ actually works. $\hat{A}$ as a set is understood in its *propositions as type* setting: it is inhabited by at least one membership function.

8.3. **General Setting for Functions.** Let $\hat{X} \vdash X$, $\hat{Y} \vdash Y$, $f : X \to Y$, and $a : X$. Suppose, furthermore, that $fa = b : Y$, $\hat{a} \vdash a$ and $\tau f = \hat{f}$. Now consider $\hat{f}\hat{a}$.

1. Either $\hat{f}\hat{a} \uparrow$ or $\hat{f}\hat{a} \downarrow$. If $\hat{f}\hat{a} \uparrow$ then $\hat{b}$ does not exist. If we assume $\hat{f}\hat{a} \uparrow$ is undecidable then nothing more can be said; if decidable, then it is reasonable to put $\hat{f}\hat{a} = \perp$.
2. Now assume $\hat{f}\hat{a} \downarrow$. We can write $\hat{f}\hat{a} = \hat{b}\langle\beta\rangle$ and $\hat{Y}\hat{b}\langle\beta\rangle = true$. This means that $\hat{f}\hat{a} = \hat{b} \vdash fa = b$. if $b \in \tau'\hat{b}\langle\beta\rangle$.

Unfortunately, this is not the end of the story. Let's look at the subset relation. It is clear that we can have subsets $X' \subseteq X$ that do not properly map elements in X' and its complement so as to preserve the identity relation.

8.4. **Guards and Approximations.** Guards represent restrictions on domains, also the meaning of the *if-then-else* statement. In effect guards merely restrict but they incur much baggage.

Suppose we invent a new idea in constructive space with the form *expr?expr* with then semantics

$$p?q = \begin{cases} p = true\ then\ value(q) \\ \perp \end{cases}$$

Now consider the definition for '$\vdash$'. We now truly introduce subset relations, but slightly different than the usual. Let $X' > A$ be a subset X' such that (i) X' is the union of all $\tau'\tau a$ for $a : X'$ and (ii) $X' \supset A$. Define $X'' < A$ similarly. then X' and X'' are both *approximations* to A.

It is clear what '$\leq$' and '$\geq$' are. Therefore, we are guaranteed that '?' has meaning. So to does the condition where both $X' \geq A$ and $X'' \leq A$.

8.5. **Approximation and Topology.** What does it mean to *approximate* in the Scott sense? There are several ways to think about it. I have made a case for constructive sets because I believe this is a proper model for examining programs.

Definition 8.2. *Constructive set A* approximates *set B is B is a constructive subset of A.*

This definition is motivated by the Kuratowski Closure Axioms:

Let X be a non-empty set and let † be an operation that assigns to each subset $A^\dagger$ of X satisfying the following axioms:
$\mathbf{K_1}$ $\emptyset^\dagger = \emptyset$.
$\mathbf{K_2}$ $A \subset A^\dagger$.
$\mathbf{K_3}$ $(A \cup B)^\dagger = A^\dagger \cup B^\dagger$.
$\mathbf{K_4}$ $(A^\dagger)^\dagger = A^\dagger$.
Then there exists one and only one topology $\mathcal{T}$ on X such that $A^\dagger$ will be the $\mathcal{T}$-closure of a subset A of X.

This is just one of many equivalent ways of defining a topology (including ultafilters). But what is † supposed to be? Obviously, it is the program and the sets are states.

What is the advantage of this formulation?

1. There is no distinction between functions and relations. In a constructive system, there is very little difference, anyway.
2. There is no distinction between deterministic and non-deterministic processing.
3. There is a natural analog between computation as done in computer science and computation as done in mathematics.
4. There is a natural analog between the categorical view and the constructive view.

Therefore, the completely presented algorithm has, as its operator, the set-building procedure that operates on the witness data and has, as the proof, a procedure that generates the preset that is the answer. With some rewording, this explanation suffices for the axiomatic semantics definition as well.

9. Conclusions

My conclusions are of two categories: (i) what can I say about the answer to the question posed in the title? and (ii) what general conclusions can I draw about foundations in computer science?

9.1. **What Is Computational Knowledge?** At its best, computational knowledge is the understanding of how to computationally answer a problem. This occurs at many levels of detail. The requirement that a *fully presented algorithm* carry its formal definition and its proof allows me to move up and down a "definiteness" scale. When programming in a concrete language, I can replace intensional understanding by extensional objects; abstraction is the opposite.

All this knowledge is encompassed by the metalanguage that I have in my head — it is psychological in nature. However, my understanding is in terms of my mental models which is indeed just my metalanguage.

9.2. **How Do We Acquire Computational Knowledge?** Acquiring computational knowledge, then, is the extension of my current models as expressed in my metalanguage. This means that I must first expand that metalanguage to accommodate new concepts. At first these concepts are fuzzy and ill-formed. Through experience and experiment I solidify the metalanguage changes. Roentgen said, "Something must be believed before it is seen." — belief in this context means that the concept must be expressible in the metalanguage.

Consider what happens to an undergraduate today. For the first several years, they are fed a steady diet of sequential thinking on a diet of von Neumann machines. They walk through the door in graduate school and suddenly that metalanguage no longer suffices. I submit that parallel and distributed computing is not within the sequential metalanguage and that language must be drastically updated. What should we do? Is parallel thinking so hard that we cannot get it to them sooner?

9.3. **General Comments.** There are several conclusions that we can draw:

1. Discovery comes first and this is primarily a psychological phenomenon.
2. The purpose of formalization is insight. In the many years since undergraduate school, I have yet to prove a theorem for the fun of it. One has a certain joy at the intellectual challenge — but rarely, if ever, is the need for insight absent.
3. There are two types of truth in mathematics: the no counterexample version and the derivability version. I propose, and there are plenty of supporting cases, that truth by construction is a valid and important criterion. See, for example, work done in safety-critical formal methods.
4. Foundations are not built on technology.

Where do we go from here? I propose that we develop a new computability quest. This new computability theory should mirror the development of analysis techniques.

1. There are many standard models in classical computability. These come over but the analysis is different. In the new theory, the question is, "What are the problems these models solve?"
2. Looking at software engineering *vis-à-vis* engineering models, we see a dearth of methods for analysis. For example, consider the table of contents of *Mathematics Reviews*: there's not a whole lot about calculus, *per se*. But there is a lot of understanding of standard models and how to analyze them.
3. We need more application schemata. This combines the above two points in the same way that differential equations models dynamics.
4. We certainly can use tools at the practioner level. Tools are developed when application schemata become important enough.
5. We must be able to reason about our models at the same level as an undergraduate engineer can reasons about the systems they work with.

References

[1] Michael J. Beeson. *Foundations of Constructive Mathematics.* Springer-Verlag, 1985.
[2] Errett Bishop and Douglas Bridges. *Constructive Analysis.* Springer-Verlag, 1985.
[3] Errett A. Bishop. *Foundations of Constructive Analysis.* McGraw-Hill, 1967.

[4] Errett A. Bishop. Mathematics as a numerical language. In A. Kino, J. Myhill, and R. E. Vesley, editors, *Intuitionism and Proof Theory*, pages 53–71. North Holland, 1967.
[5] Errett E. Bishop. The crisis in contemporary mathematics. *Historia Mathematica*, 2:507–517, 1975.
[6] Errett E. Bishop. Schizophrenia in contemporary mathematics. *Contemporary Mathematics*, 39:1–32, 1985.
[7] Errett J. Bishop and Douglas Bridges. *Constructive Analysis*. Springer-Verlag, 1985.
[8] Noam Chomsky. Three models for the description of language. *IEEE Trans. on Information Theory*, 2(3):113–124, 1956.
[9] R. L. Constable and *et al. Implementing Mathematics with the NuPrl Proof Development System*. Prentice-Hall, 1986.
[10] S. Feferman. Constructive theories of functions and classes. In M. Boffa, D. van Dalen, and K. McAloon, editors, *Logic Colloquia 1978 Mons*, number 97 in Stud. Logic Found Math., pages 159–224. North Holland, 1979.
[11] James H. Fetzer. Program verification: the very idea. *CACM*, 31(9), 1988.
[12] Stephen J. Gould. *Full House*. Random House?, 1995.
[13] Carl A. Gunter. *Semantics of Programming Languages: Structures and techniques*. MIT Press, 1992.
[14] Fred Hennie. *Introduction to Computability*. Addison-Wesley, 1977.
[15] S. C. Kleene. *Introduction to Metamathematics*. North Holland Publishing Company, 1971. Original 1952.
[16] A. Kolmogoroff. Zur Deutung der intuitionistischen Logik. *Mathmatische Zeitschrift*, 33:58–65, 1932.
[17] Thomas S. Kuhn. *The Structure of Scientific Revolutions*. University of Chicago Press, Chicago, IL, 2 edition, 1970.
[18] John McCarthy. A basis for a mathematical theory of computation. *Computer Programming and Formal Systems*, pages 33–70, 1963.
[19] Lawrence C. Paulson. *Logic and computation : interactive proof with Cambridge LCF*. Cambridge University Press, 1987.
[20] Aarne Ranta. *Type-Theoretical Grammar*. Clarendon Press, 1994.
[21] W. Rudin. *Real Analysis*. Macmillan, 1968.
[22] D. E. Stevenson. Science, computational science, and computer science: At a crossroads. *Comm. ACM*, 37(12):85–96, 1994.
[23] D. E. Stevenson. Principles of constructive euclidean geometry. *Bulletin of the AMS*, Submitted.
[24] Patrick Suppes. *Introduction to Logic*. Van Nostrand, 1971.
[25] A. S. Troelstra. *Principles of Intuitionism*. Number 95 in Lecture Notes in Mathematics. Springer-Verlag, 1969.
[26] A. M. Turing. On computable numbers, with an application to the "entscheidungsproglem". *Proc. London Math. Soc. Ser 2*, 42:230–265, 1935.
[27] Friedrich Waismann. *The principles of linguistic philosophy*. Macmillan, 1965.
[28] Niklaus Wirth. *Algorithms + data structures=programs*. Prentice-Hall, 1976.

442 R. C. Edwards Hall Department of Computer Science, Clemson University, PO Box 341906, Clemson, SC 29634-1906

E-mail address: steve@cs.clemson.edu