WHAT IS A MODEL?

A CONSUMER'S PERSPECTIVE ON SEMANTIC THEORY

by

Jon Shultis

CU-CS-303-85 July, 1985

University of Colorado, Department of Computer Science,
Boulder, Colorado.

What is a Model?
A Consumer's Perspective on Semantic Theory

Jon Shultis
Department of Computer Science
University of Colorado
Boulder, CO 80309

Abstract

Much of semantic theory, as currently practiced, is of little use to the working programming language designer and implementor. Language designers want constructive methods for semantic modelling, so that the semantics provides mechanical design assistance. We argue that the gap between theory and practice can be bridged by adherence to a simple principle, viz. syntax should express constructions in the model, and nothing more.

WHAT IS A MODEL?
A CONSUMER'S PERSPECTIVE ON SEMANTIC THEORY

Jon Shultis
Department of Computer Science
University of Colorado
Boulder, Co 80309, USA

Introduction

By a "consumer" of semantic theory, I mean anyone who, like myself, is not primarily concerned with proving new results or otherwise contributing directly to the theory of semantics, but who finds (or could find) the concepts and methods of semantics useful for doing other work. Currently, consumers include programming language designers and, to some extent, language implementors. The results and methods of semantics could serve a much wider audience.

Insofar as everyday programming is a process of (language) specification and implementation, why should semantics seem so irrelevant to programmers? Whenever I treat semantic theory in my graduate programming languages class, the students listen patiently for about a week, and then begin asking: "What is this stuff good for?" I am getting better at finding answers to pacify my students, but the question nags at me more all the time. Take money, for example. How many cases can we point to where a result from semantic theory has saved someone a penny? Not one.

A thing is useful if it solves a problem that someone wants solved. So far, semantics has solved problems that are of interest mainly to semanticists: How can this kind of language be modelled? When are two programs equivalent? What do all models of this language have in common? When does a language have a relatively complete theory? And so forth. The answers that have been found for these and many other questions have unquestionably improved our understanding of the

basic issues and methods of semantic theory.

Should I then tell my students that these concepts and techniques are good for doing more semantic theory? In other words, "this stuff is good for solving problems that most of you don't care about". If that's the best I can do, I shouldn't be teaching semantics to computer scientists.

Semantic theory can be far more relevant and useful to the practically-oriented computer scientist than it is at present. To see how, we need to take a careful look at the kinds of problems faced by consumers of semantics, and understand why the theory sometimes helps solve those problems, and sometimes doesn't.

1. Metalanguage Semantics and Constructivity

The controversy and confusion over how to interpret the definition of Algol 60 posed a problem that computer scientists wanted solved: some means had to be found of defining the meaning of a programming language in a precise and unambiguous way.

Strachey's solution [21] was to associate a mathematical denotation with each phrase of a language. In order to do this, he faced a subsidiary problem: how to write down the denotations, and how to write down the mapping from language structures to their denotations. He settled on Church's λ-notation for writing down the denotations, and syntax-directed translation for writing down the mapping. A third problem, noticed by Scott, is that the metalanguage used to specify the semantics also needs a semantic definition; reflexive domains provided the solution for λ-calculus. The method of denotational semantics is now routinely used to define small languages and study their properties. Several examples of such

applications are contained in these proceedings.

The similarity between the specification of a denotational semantics and a model has led to the identification of the two. Recall that, in mathematical logic, a domain together with a homomorphism from the syntax to that domain is called an *interpretation*. Given a notion of truth in the domain, an interpretation is a *model* of a logic (theory) if all of the axioms are true and all of the inference rules preserve truth. In the case of programming languages, we can think of a denotational semantics as giving a model of the language's Floyd-Hoare theory. In the case of λ-calculus models, we might use realizability as our notion of truth, in which case all that really needs to be checked is that the conversion rules are valid.

The identification of denotational semantics with models is unfortunate, however, because it misses a point of paramount importance to consumers. The success of denotational semantics is largely due to the fact that the λ-calculus semantics we use is constructive, in the sense that writing down the semantic equations for a language gives an effective way of computing the meaning of any phrase.

To illustrate the importance of this point, consider the denotational semantics for Communicating Sequential Processes given by Brookes et al. in [5]. The metalanguage used there is the language of classical first-order logic and set theory. Ignoring the philosophical issues, one thing is clear: whatever semantics we give to this metalanguage, it will not be effective. There is nothing "wrong" with this, unless we want to use the semantics as a basis for an implementation, or use some theorems about the semantics to manipulate actual programs. Because these are precisely the things that consumers want semantics for, however, a non-effective semantics is less useful to them than it might be.

Among other things, the non-effectiveness of the semantics causes trouble when operations for combining processes are defined. For example, Brookes *et al.*'s Theorem 2 asserts that the intersection of an arbitrary family of processes is a process. In order to understand why this is so from a computational standpoint, one needs a proof that yields a construction of the intersection, showing why the result is again a process. Unfortunately, the proof offered uses the method of contradiction, which is not effective. The burden of devising a constructive proof is therefore laid squarely on the shoulders of anyone who might wish to use the result as part of a computer program. Since there is no guarantee *a priori* that such a proof can be found, the "result" might just as well have been stated as a conjecture.

By contrast, consider the theorem that for every nondeterministic finite automaton there is an equivalent deterministic one. The usual proof of this gives a direct construction of the deterministic automaton from the nondeterministic one, and shows that the construction leaves invariant the language accepted. The theorem is useful to consumers of automata theory because its proof supplies an algorithm.

By analogy, denotations (for programming semantics, at least!) should be effective objects, effectively given, and effectively reasoned about. In short, a denotational semantics should provide an *effective* model, not just any model. For a discussion of effectiveness, see [24]. Adherence to this principle would go a long way toward making semantic theory more useful. But is it enough?

2. What is a Model?

Consider the category $\mathbf{N}\leq$ having the natural numbers as objects, the arrows being given by the usual ordering. It is easy to model addition using $\mathbf{N}\leq$ as the

domain. Let the numeral n denote the corresponding arrow $\hat{n}:0 \leq \overline{n}$, where $\overline{n}$ is the (semantic) natural number corresponding to the (syntactic) numeral n. The semantics of addition is given by the following equation.

$$[\![n+m]\!] = [\![n]\!] \circ (trans([\![m]\!], \overline{n}))$$

where

$$trans:(\overline{i} \leq \overline{j}, \overline{k}) \vdash\!\!\rightarrow (\overline{i+k}) \leq (\overline{j+k})$$

Imagine now that we have been given a machine (the N≤ machine) with a screen that displays two numerals separated by the symbol "≤", and several buttons. There is a button for each digit, and buttons labelled "from", "to", and "compose". To operate the machine, we type in a sequence of digits, followed by "from", followed by a second sequence of digits, followed by "to". The digit sequences are displayed on either side of the "≤" on the screen. Next, we press "compose", then enter a second pair of digits. After the last digit has been entered and the "to" button pressed, one of two things happens. If the second sequence of digits of the first pair is the same as the first sequence of the second pair, then the first sequence of the first pair and the second sequence of the second pair are displayed on the screen. Otherwise, the screen displays the word "error".

The semantics given above tells how to model the language of addition expressions in N≤, but does this mean that we can use the N≤ machine as an adding machine? Clearly not. The semantics is certainly effective, but it depends on a construction, viz. *trans*, that has no counterpart in N≤ (and hence is not an operation of the machine).

The example of the N≤ machine illustrates a common source of frustration with the methods of denotational semantics. Programmers constantly have to define the semantics of languages (i.e., software interfaces) in terms of (hardware

and software) devices with limited capabilities, such as disk controllers, report generators, and so forth. For such purposes, the fact that the device is an effective model in some general sense of "effective" is not enough. The semantics has to be effective *with respect to the capabilities of the device.*

If we restrict ourselves to general-purpose programming languages, this problem is never apparent, as long as we use λ-calculus or some other constructive notation for our metalanguage. In many situations, however, it is important to recognize the problem and avoid it by adhering to the following principle: the semantics of a language should use constructions from the semantic domain(s) of the model, and nothing else. In the parlance of topos theory, the semantics should be defined entirely in terms of the internal structure of the domains.

Insofar as a model is supposed to capture the "meaning" of a language, shouldn't we always avoid shifting any of that meaning into a metalanguage the semantics of which is not explicitly represented in the model? Perhaps we should revise our definition of "model" to include the metalanguage somehow. In order to avoid confusion, however, we shall continue to use "model" in its conventional sense, and we shall refer to internally constructed models as "internal models".

To a large extent, interest in categories that admit universal objects [23] is motivated by the need to consider the semantics of the metalanguage when developing the semantics of a subject language. A universal domain, coupled with a language in which to express constructions in that domain (as in, e.g., ML [10]), allows one to focus on the details of a specific subject language (or, more generally, program). The problem of metalanguage semantics can be ignored because the metalanguage is fixed and predefined. Of course, this is exactly what we do when we use λ-notation and domain equations to give semantics to ordinary sequential

programming languages.

Learning the structure of a universal domain such as Pω [22] requires a significant investment of time and effort. Unless the return on that investment is a great increase in practical problem-solving ability, consumers will not be motivated to make it. If the metalanguage is a powerful problem-solving tool, they are more likely simply to use it, and take the theory behind it for granted.

As with a first programming language, however, one's ability to conceive of alternative solutions to a problem may be limited by taking the universal domain for granted. This problem can become acute if the universal domain is inappropriate to a problem, e.g. Pω is inadequate for studying nondeterministic programs.

Suppose that, instead of a specific universal domain construction, the consumer were offered tools for effectively constructing (universal) domains, and shown some powerful applications of those tools. Such tools would make the study of domain constructions more appealing to consumers, because the results of such constructions would be perceived as having immediate practical application. We will return to this point in §4.

3. Cartesian Closure and λ-calculus Models

How does restricting attention to internal models affect the results of semantic investigations? To illustrate the difference between internal and external models, we reconsider Albert Meyer's question, "What is a Lambda-Calculus Model?" [19], taking the work of Berry [3] as a starting point.

Berry's thesis is that "*the cartesian closure is really the key property for semantic model constructions*" (Berry's emphasis). The main result supporting this

claim is his theorem 5.2.9, which states that "Any categorical model defines a model of Λ." (We shall explain these terms presently.) To what extent is this thesis upheld when we replace "model" by "internal model"?

We begin by summarizing some definitions. (Caveat: we write both composition and application in diagrammatic order! So, for example, xf denotes f applied to x, which we prefer to enunciate as "x supplied to f". This perversity is the sorry result of having been brought up to read programs from left-to-right and top-to-bottom.)

Λ is the syntactic domain of λ-expressions. A *model of* Λ consists of the following data.

- Three domains D, V, Env, of denotations, values, and environments, respectively.
- A *semantic functor* $[\![\,]\!]:\Lambda \to D$
  $[\![\,]\!]$ must preserve (syntactic) α- and β-conversion,
  and semantic equivalence must imply syntactic substitutivity.
- An evaluation operator $eval:Env \times D \to V$
- An application operator $\bullet: V \times V \to V$

The environment domain is defined by the equation $Env = \prod_{\omega} V$. When no confusion is possible, we write ρd for $(\rho,d)eval$. These data must satisfy the following conditions.

$$(var) \qquad \rho[\![x]\!] = \rho\pi_x$$

$$(app) \qquad \rho[\![ee']\!] = (\rho[\![e]\!], \rho[\![e']\!])\bullet$$

$$(lambda) \qquad (\rho[\![\lambda x.e]\!], v)\bullet = (\rho[v/x])[\![e]\!]$$

$$(free) \qquad \rho|_{fv(e)} = \rho'|_{fv(e)} \Rightarrow \rho[\![e]\!] = \rho'[\![e]\!]$$

where $\rho|_{fv(e)}$ denotes the projection of ρ on the set of free variables of e.

A *categorical model* is given by the following data.

- A Cartesian-closed category **C** with terminal object $\top$ and evaluation arrow *ev*. f denotes the exponential adjoint of f.
- $V_{\mathbf{C}} \in Ob(\mathbf{C})$
- A retraction $\Theta: V_{\mathbf{C}}^{V_{\mathbf{C}}} \to V_{\mathbf{C}}$ with Θ^{-1} the corresponding section
- $Env_{\mathbf{C}} = \prod_{\omega} V_{\mathbf{C}}$.

Updating in $Env_{\mathbf{C}}$ is accomplished by the collection of arrows $\{s_x: Env_{\mathbf{C}} \times V_{\mathbf{C}} \to Env_{\mathbf{C}} \mid x \in \omega\}$, one for each "variable" x, defined by the equations $s_x \circ \pi_x = \pi_2$, and $s_x \circ \pi_y = \pi_1 \circ \pi_y$ for $x \neq y$.

We now attempt to prove that from every categorical model a model of Λ can be constructed *internally*. The problem is: internal to what? The statement that **C** is a Cartesian-closed category implies, for example, a bifunctor $\pi: \mathbf{C} \times \mathbf{C} \to \mathbf{C}$ which is a binary Cartesian product on **C**. Berry makes free use of π, which is external to **C**, but internal to the world in which **C** is defined.

Berry's thesis is ambiguous on this point; is the key property the property of *being* a Cartesian-closed category (with some additional structure), or is it the property of *containing* such a category? If we choose the second interpretation, then Berry's proof suffices. In that case, we interpret the theorem as saying that "any domain in which a categorical model can be defined is an internal model of Λ". If, on the other hand, we choose the first interpretation, then we need a new proof, and interpret the theorem as "every categorical model is an internal model of Λ".

Since there is nothing to do otherwise, we shall choose the first interpretation, and see how far we get. To start, we must choose objects of **C** for the domains. (For convenience, we shall henceforth drop the subscript **C** from the names of **C**-objects where no confusion is possible.) For denotations we take V^{Env}, for values we take V, and for environments we take Env.

For *eval* we take $ev{:}D \times Env \to V$. The restriction of *ev* to domain $D \times Env$ emphasizes that *eval* is a C-morphism, and so is internal to C; we do not at any time assume that the full unit of the exponential adjunction is available when constructing meanings. For $\bullet$ we take $(\Theta^{-1} \times 1) \circ eval$, also a C-morphism. It is vital that both of these be defined in advance as fixed C-morphisms, so that when they are used later in the semantic clauses no external constructions are inadvertently introduced.

The semantic clauses must associate with each syntactic phrase an element $\phi{:}\top \to V^{Env}$ of the semantic domain. ϕ must be specified either by a fixed C-morphism, or as a composition of such morphisms. The product and exponentiation functors are part of the external description (or specification) of C and so cannot be used except as a means of naming a particular C-morphism as in our definition of $\bullet$ above. We start with the semantics of variables.

$$[\![x]\!] = \pi_x$$

Notice that the use of exponentiation is apparently allowed under the rules cited above, but we will find cause to object to it later. The "obvious" semantics of application is expressed as follows.

$$[\![ee']\!] = (\Delta_{Env} \circ (((!_D \circ [\![e']\!]) \times 1_{Env}) \circ eval \circ \Theta^{-1}) \times ((!_D \circ [\![e]\!] \times 1_{Env}) \circ eval)) \circ eval)$$

where $\Delta_{Env}{:}Env \to Env \times Env$ is the diagonal on *Env*, $!_D{:}D \to \top$ is the unique morphism, and 1_{Env} is the identity on *Env*, all of which are fixed C-morphisms. However, the clause makes explicit use of both exponentiation and product, which is disallowed. Though we have no formal proof to offer, we believe that they cannot be eliminated, in which case Berry's thesis is denied internally.

One could argue that the clause is merely a finite presentation of an indefinite sequence of associations of specific syntactic phrases to specific C-morphisms, and

as such the use of the functors should be allowed. After all, isn't the use of external functors in this way implicit in the semantics of variables? It is; hence our objection to that clause. Berry considers the closure of **C** under ω-products a mere convenience, stating that "...it simply avoids counting variables...". If we wish to have effective models, we must interpret ω as representing the process of counting, not a completed whole. If we wish to have counting internally, we must have a natural numbers object in **C**. In short, counting variables is something we cannot really avoid!

We conclude that the second interpretation of Berry's theorem is the correct one, viz. that every domain in which a categorical model can be defined is an internal model of Λ. For the consumer, this means that if a device is capable of defining a categorical model, then it is capable of doing λ-calculus. A device which merely is a categorical model could be used by some more powerful device to represent the denotations of λ-expressions, but is not itself capable of being a λ-calculator.

Incidentally, the converse of Berry's theorem is also true; a categorical model can be defined in every internal model of Λ. The easiest way to prove this is to construct a categorical model using pure λ-calculus. The necessary λ-expressions for pairing, projection, exponentiation (Currying), evaluation, etc., can be found in any standard treatment of λ-calculus, e.g. [1].

4. Domains for Doing Semantics

Our program for making semantic theory more relevant to computing requires us, first, to make explicit the domain in which any construction is being carried out and, second, to ensure that all constructions (proofs) are effective. In this section

we sketch a plan for effectively realizing these goals. In particular, we consider the design of a universal metalanguage within which to do semantic theory.

Semantics draws on many branches of mathematics: algebra, topology, set theory, logic, category theory, and so forth. Any language for doing semantics must therefore be capable of expressing concepts and results in all of these domains, as well as others that have not yet been invented. All of these areas share three things, however: the ability to define new structures, to quantify over those structures, and a set of basic rules for reasoning about them. Using these three tools - definition, quantification, and logic - all of modern mathematics is built up from some initial theory, e.g. set theory.

These three ingredients for the synthesis of mathematics therefore serve also as the basic ingredients of our language. Definition serves to specify new domains (externally), quantification serves to limit consideration to those domains (i.e., to their internal structure), and logic is the key to effectiveness.

Higher-order intuitionistic logic (or "intuitionistic type theory") provides all three ingredients. The use of intuitionistic logics for programming has been studied by several authors, notably [18], [20], [7]. Since we are concerned here primarily with semantic theory, and because categorial methods are becoming more popular in the semantics community, we are interested in finding a version of intuitionistic logic which can provide an effective basis for doing category theory in a straightforward way.

We therefore take as our basic system an axiomatization of the logic of partial elements, due to Fourman and Scott [8], which we will call **FS**. The syntax of the terms and formulae of **FS** is given by the following grammar.

$$<term> ::= <var> \mid <const> \mid !<var><formula>$$

$$<formula> ::= <var> \mid \mathrm{E}<\mathrm{term}> \mid <\mathrm{term}> \equiv <\mathrm{term}>$$
$$\mid <term>(<term>, \cdots, <term>) \mid <formula> \wedge <formula>$$
$$\mid <formula> \rightarrow <formula> \mid \forall <var> <formula>$$

The logic is couched in terms of schemata for the axioms and inference rules rather than actual formulae. A schema is just like a term or formula except that certain subterms and subformulae are replaced by metavariables standing for arbitrary terms or formulae.

As it happens, it is easier to give semantics to these schemata than it is to restrict ourselves to actual terms and formulae. In fact, instantiation of a schema has no effect on the "code" that realizes that schema, because the code given for the schema always works, for all instances; it is "universal".

Because we want proofs to be effective, we are primarily interested in *realizability* semantics [13] The important difference between our semantics and, for example, the **Ω-Set** semantics originally proposed by Fourman and Scott is that the latter lacks "computational content" (D. Scott, in response to my question posed at the conference). In other words, the **Ω-Set** semantics is precisely the kind of semantic theory which is of no use to consumers.

Here, the realization of a formula (schema) is (the program denoted by) some λ-expression. For concreteness, we take for our λ-calculus semantics Berry and Curien's sequential algorithms on concrete data structures [4]. An alternative would be to calculate a Gödel number from the λ-expressions, giving a more traditional realization, along the lines of Scott's recent work.

Certain formula schemata (the axioms) are given realizations *ab initio*. The inference rules transform formula schemata into new formula schemata, and their

realizations into corresponding new realizations. The axiom schemata are listed below. The name of each schema is given first, followed by the formula, with the realization written on the following line.

4.1. Propositional Axioms

$\mathbf{K}: \vdash \phi \to (\psi \to \phi)$

$\lambda x.\lambda y.x$

$\mathbf{S}: \vdash (\phi \to (\psi \to \theta)) \to ((\phi \to \psi) \to (\phi \to \theta))$

$\lambda x.\lambda y.\lambda z.(x\ z)(y\ z)$

$\pi_1: \vdash (\phi \wedge \psi) \to \phi$

$\lambda(x,y).x$

$\pi_2: \vdash (\phi \wedge \psi) \to \phi$

$\lambda(x,y).y$

$pair: \vdash \phi \to (\psi \to (\phi \wedge \psi))$

$\lambda x.\lambda y.(x,y)$

The propositional axioms define a quasi-Cartesian-closed structure that is internal to the logic. **K** specifies the existence of all exponent objects, and **S** defines a kind of internal evaluation functor; the last three axioms specify the binary projections and pairing functor.

4.2. First-Order Axioms

Some preliminary remarks may help to clarify the semantics of the first-order axioms. Notice that all computation is essentially propositional. In particular, it is well-known that the combinators **S** and **K** are a sufficient basis for the partial

recursive functions. First-order logic deals with the concept of *element*, thereby adding an infrastructure to the propositions. In terms of programs, first-order statements "merely" allow us to give more detailed interpretations to what are essentially propositional combinations.

In order to speak of elements, we must postulate for them a source, which we shall denote by "()" (or "*nil*"). () has the property that there is a unique arrow from any object to () (i.e., it is terminal). In the category of sets, for example, any singleton can be used as (). An *element* is defined as any arrow from () to any other object. Again, in the category of sets, we can identify the "elements" of any set S with the collection of functions $e{:}() \to S$. In keeping with categorial terminology, we shall refer to arrows with domain () as elements. The "value" of an element e, denoted by $e()$, will be called an *item*.

Conceptually, the formula $\tau \equiv \sigma$ represents an equivalence class of items. The realization of such an equivalence class will be a representative item. If $\vdash \sigma \equiv \tau$, then σ and τ denote indistinguishable items. We are now ready to present the first-order axioms.

substitutivity: $\vdash \phi[y/x] \wedge y \equiv z \to \phi[z/x]$

$\lambda(f,i).f$

Given that y and z are indistinguishable, a realization of $\phi[y/x]$ is equally satisfactory as a realization of $\phi[z/x]$.

extensionality: $\vdash \forall x(x \equiv y \leftrightarrow x \equiv z) \to y \equiv z$

$\lambda e.e()$

Given an element e, we can produce the corresponding item $e()$.

instantiation: $\vdash \forall x\phi \wedge \mathbf{E}x \to \phi$

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

This is just the law of functional application (essentially the evaluation functor), which asserts that a function can be applied to any existing element (of the appropriate sort - **FS** is a many-sorted logic, but the details of sorts are not needed for the current discussion. The interested reader is referred to [8].)

description: $\vdash \forall y(y \equiv !x\phi \leftrightarrow \forall x(\phi \leftrightarrow x \equiv y))$

$\lambda y.(\lambda e.\lambda x.e,\ \lambda x.x())$

This axiom asserts that we can convert freely between the equivalence class of an item specified uniquely by the formula ϕ and the corresponding element.

4.3. Higher-Order Axiom

comprehension: $\vdash \mathbf{E}!y\ \forall \overline{x}(\phi \leftrightarrow y(\overline{x}))$

$\lambda \overline{x}.()$

Higher-order logic deals with collections of elements, or "sets". The axiom of comprehension asserts that all specifiable collections exist or, more precisely, that all structures have characteristic terms (which can then be quantified over). The justification for this is that $\lambda \overline{x}.()$ exists. That is, we specify the unique map from any collection (including the empty collection) to (), the "terminal object"; everything else follows.

Incidentally, the original formulation of **FS** included an additional axioms of *predication*. Its effect is to give the logic strict semantics for predication. We do not include it here, however, because in order to realize it we would need to retain the entire history of a computation so that from any result we could recover the operator and operands which produced it. In practice, the absence of the axiom of predication is hardly noticed.

4.4. Inference Rules

There are four inference rules, each of which transforms formula schemata and their realizations into new formula schemata and realizations. In the rules, the realization of a formula is written beside it in set braces, "{ }".

4.4.1. Modus Ponens

$$\frac{\phi\ \{x\} \quad (\phi \to \psi)\ \{f\}}{\psi\ \{fx\}}$$

4.4.2. Curry (∀-introduction)

$$\frac{\psi \wedge \mathrm{E}x \to \phi\ \{f\}}{\psi \to \forall x \phi\ \{(\lambda x.\lambda y.f(x,y))\}}$$

(x not free in ψ)

4.4.3. Substitution

$$\frac{\phi\ \{f\}}{\phi[\tau/x]\ \{f\}}$$

(x a term variable)

4.4.4. Elaboration

$$\frac{\phi\ \{f\}}{}$$

$$\phi[\psi/x]\ \{f\}$$

(x a formula variable)

Fourman shows how **FS** defines a free (elementary) topos $E()$. A topos is a Cartesian-closed category with a subobject classifier - an object Ω with the property that for every subobject $s:a \rightarrow b$ there is a unique morphism $\chi_a:b \rightarrow \Omega$ such that $\chi_a \circ s$ factors uniquely through $\top$ (i.e., *nil*). In terms of our semantics, the factorization is given by $\mathbf{true} \circ (\lambda \overline{x}.())$, where **true** is a distinguished element of Ω, the domain of "truth values".

Elementary topoi were discovered by Lawvere and Tierney in the late 60's (see [14] [15]). The early development of the theory, and its relationship to intuitionistic logic, is recorded in conference proceedings from the early 70's, especially [16], [17]. A good introduction to topos theory, which concentrates on the relation of topoi to logic, is Goldblatt [9]. A more technical treatment is given by Johnstone [11], but is less accessible to the non-specialist. MacLane has recently completed a book on the subject, as well, but I do not have an exact reference for it, nor have I seen it.

Elementary topoi are essentially versions of set theory. The fact that $E()$ is free in the "category" of elementary topoi means that there is a unique functor from $E()$ to any other topos; it is the least constrained set theory there is. Since the semantics of **FS** is effective, $E()$ is an effective domain. By building up mathematics, especially semantic theory, within $E()$, we can guarantee (Church-Turing) effectiveness. To do this, we can use the logic **FS** itself, or we may prefer a more purely categorial, but equivalent, notation.

An enhanced version of our **FS** realizability model is incorporated in a computer program called **Intuit**. **Intuit** supports natural deduction proofs, and actu-

ally works on a syntactic representation of the realization of a formula, which it optimizes extensively. A small library of theorems has been compiled, but so far no significant theories have been developed.

We must emphasize that a great deal of groundwork has to be done before we could hope even to start doing semantic theory with the **Intuit** system. An enormous body of mathematics has first to be reconstructed essentially "from scratch", a project of essentially the same scope and magnitude as Bourbaki, requiring contributions from many mathematicians over a period of decades. We envision a massive on-line library of definitions, theorems, and proofs, to be a repository of constructive mathematical knowledge in the public domain. Any qualified person could contribute to the library or draw from it for further work. If programming in this manner became sufficiently popular, one might even expect to see the formation of private libraries of proprietary theorems and proofs that can be used for a fee.

A major problem for such a project would be the organization of the library of definitions, theorems, and proofs into a coherent and easily-referenced whole. The PRL systems [2] support an organization based on simple lists of theorems, functions, definitions, and some primitive recursive utilities. This falls far short of their goal (inspired by Kenneth Wilson) of a "library of results organized into books, chapters, sections, and so on." We sketch one possible alternative organization in the following section.

5. Branches and Categories

Conventional libraries are surprisingly inefficient databases. They are highly redundant, badly cross-indexed, and do not support shared access to information.

Much of the information is out-of-date, difficult to place in its proper context, and the relationships among data that are perceived by the expert in an area are nowhere apparent. Perhaps worst of all, knowledge is packaged by author, rather than by topic. We therefore reject the paradigm of a conventional library, and consider instead an organization of mathematical knowledge into branches, topics, results, and proofs (to which authorship may be appended as a footnote). At the highest level, all of the branches are organized under category theory, which captures knowledge about relationships among the various branches. (Books, written by various authors for a variety of approaches and styles, will of course continue to be useful as guides to using the library.)

As mentioned earlier, **FS** provides an effective language for doing "free set theory" (actually, topos theory). In order to work in other branches of mathematics, we need to introduce theories for them. Traditionally, a theory is defined by a set of sorts and constants, and a set of formulae (axioms). To prove a formula ϕ in a theory Γ is essentially the same as to prove the entailment $\Gamma \vdash \phi$ in the base logic. If Γ is defined as an extension to **FS** in this way and the proof uses **FS** freely, then we say that ϕ holds *externally to* Γ *within* **FS**.

We generalize this terminology in the obvious way. Given a theory Δ, a theory $\Gamma = \Delta \cup E$ is an *extension* of the *base theory* Δ. If $E \vdash \phi$ in Δ, then ϕ holds externally to E within Δ.

In order to support internal reasoning, we need to add a mechanism for encapsulating theories. To accomodate this new concept, we change the traditional definitions slightly. We define a *theory* to be any set Γ of definitions, formulae, and inference rules. A theory Δ is an *extension* of Γ if Δ includes Γ, and any new inference rules in Δ are derived from those in Γ together with the axioms of Δ. Δ

is a *branch* of Γ if $\Delta \cup \Gamma$ is an extension of Γ. The important point here is that Δ may discard ("hide") parts of Γ. To continue with arboreal terminology, **FS** is the *root* theory; all other theories are either extensions or branches of **FS** (or its extensions and branches).

The concept of branch captures the notion of "internal theory"; a branch may be isolated from its base, though it may also inherit things from the base. Any formula proved in a branch, however, gives a construction that is completely internal to that branch. (Note that topoi are not the only structures having internal theories, though one might hesitate to call the internal theory of anything but a topos an internal *logic*.)

Branches enable us to define and work within various areas of mathematics, but they don't provide any means of relating one branch to another, except as base and extension. The goal of category theory, of course, is precisely to define and study relationships and transformations among the various branches of mathematics. Moreover, the search for universals is properly conducted within category theory. If category theory is simply added as another branch to the great tree of mathematics, however, it will be isolated from all of the other branches!

The resolution of this quandary is beyond the scope of the current work, but essentially involves the notion of *indexed categories* [12]. Briefly, the theory of indexed categories is a theory about the tree of mathematics. Its objects are "indexed categories" - families of morphisms "enumerated" by some object of a base topos. Instead of working directly with branches of **FS**, we can work with those branches as defined within the branch of indexed categories. (An alternative which will suggest itself readily to language designers would be to use multiple inheritance instead of the simple tree proposed here. We do not know how to

introduce multiple inheritance in a clean way in this context, however.)

In order to avoid needless duplication, we propose to provide, as our universal domain for doing semantics, the branch of indexed category theory. In short, we propose to replace "set theory" by (indexed) category theory as the "initial object" from which all of mathematics would be derived. In this way, we can encourage all branches of mathematics to be developed in the appropriate categorial climate, so they can inherit categorial results directly instead of proving those results in each special case and later exhibiting them as "examples of the general concept".

At the outermost level, then, we would have a language for defining various categories and functors between them. Among those functors would be some basic ways of combining categories, along the lines of Clear [6]: extend, restrict, combine, etc. The categories, together with the base theory of indexed categories, would be the branches of mathematics. Knowledge within each branch would be catalogued according to topic, theorem, and proof. Beneath each branch that includes branch-growing functors would be sub-branches, with their own topics, theorems, proofs, and (possibly) sub-branches. The natural scope rules arising from this organization would effectively indicate the branch within which any construction is carried out.

References

1. H. P. Barendregt, *The Lambda-Calculus: Its Syntax and Semantics*, North-Holland, 1981.

2. J. L. Bates and R. L. Constable, Proofs as Programs, *ACM Trans. Prog. Lang. and Systems 7*, 1 (January 1985), 113-136.

3. G. Berry, Some Syntactic and Categorical Constructions of Lambda-Calculus Models, INRIA Rapport de Recherche No. 80, June 1981.

4. G. Berry and P. L. Curien, Sequential Algorithms on Concrete Data Structures, *Theoretical Computer Science*, , 1982.

5. S. D. Brookes, C. A. R. Hoare and A. W. Roscoe, A Theory of Communicating Sequential Processes, *J. ACM 31*, 3 (July 1984), 560-599.

6. R. M. Burstall and J. A. Goguen, Putting Theories Together To Make Specifications, *5th International Joint Conference on Artificial Intelligence*, , 1977, 1045-1058.

7. R. L. Constable and D. R. Zlatin, The Type Theory of PL/CV3, *ACM Trans. Prog. Lang. and Systems 6*, 1 (January 1984), 94-117.

8. M. P. Fourman, The Logic of Topoi, in *Handbook of Mathematical Logic*, J. Barwise (ed.), North-Holland, 1977, 1054-1090.

9. R. Goldblatt, *Topoi: the Categorial Analysis of Logic*, North-Holland, 1979.

10. M. J. Gordon, A. J. Milner and C. P. Wadsworth, Edinburgh LCF, in *Lecture Notes in Computer Science, Vol. 78*, Springer, Berlin, 1979.

11. P. T. Johnstone, *Topos Theory*, Academic Press, 1977.

12. P. T. Johnstone and R. Pare, eds., *Indexed Categories and Their Applications*, Springer Lecture Notes in Mathematics, 1978.

13. S. C. Kleene, On the Interpretation of Intuitionistic Number Theory, *J. Symbolic Logic 10*, (1945), 109-124.

14. F. W. Lawvere, Adjointness in Foundations, *Dialectica 23*, (1969), 281-296.

15. F. W. Lawvere, Quantifiers and Sheaves, in *Actes des Congres International des Mathematiques, tome I*, 1970, 329-334.

16. F. W. Lawvere, ed., *Toposes, Algebraic Geometry, and Logic*, Springer Lecture Notes in Mathematics, 1972.

17. F. W. Lawvere, C. Maurer and G. C. Wraith, eds., *Model Theory and Topoi*, Springer Lecture Notes in Mathematics, 1975.

18. P. Martin-L$umlautof, Constructive Mathematics and Computer Programming, *6th Int'l. Congress for Logic, Methodology, and Philosophy of Science*, Hannover, Aug. 1979.

19. A. R. Meyer, What is a Lambda-Calculus Model?, MIT LCS/TM-171, August 1980.

20. D. Scott, Constructive Validity, in *Proc. Symposium on Automatic Deduction, Lecture Notes in Mathematics 125*, Springer-Verlag, 1970, 237-275.

21. D. S. Scott and C. Strachey, Toward a Mathematical Semantics for Computer Languages, in *Proc. Symposium on Computers and Automata*, J. Fox (ed.), Polytechnic Institute of New York, New York, 1971, 19-46.

22. D. Scott, Data Types as Lattices, *SIAM J. Comput. 5*, 3 (Sept. 1976), 522-587.

23. M. B. Smyth and G. D. Plotkin, The Category-Theoretic Solution of Recursive Domain Equations, *Proc. 18th Symposium on Mathematical Foundations of Computer Science*, , 1977, 13-17.

24. M. B. Smyth, Effectively Given Domains, *Theoretical Computer Science 5*, (1978), .