This paper appears in *Mathematical Structures in Computer Science*, volume 1, number 1, pages 49–67, March 1991.

A Categorical Manifesto[1]

Joseph A. Goguen
Programming Research Group, Oxford University
SRI International, Menlo Park CA

Abstract: This paper tries to explain why and how category theory is useful in computing science, by giving guidelines for applying seven basic categorical concepts: category, functor, natural transformation, limit, adjoint, colimit and comma category. Some examples, intuition, and references are given for each concept, but completeness is not attempted. Some additional categorical concepts and some suggestions for further research are also mentioned. The paper concludes with some philosophical discussion.

0 Introduction

This paper tries to explain why category theory is useful in computing science. The basic answer is that computing science is a young field that is growing rapidly, is poorly organised, and needs all the help it can get, and that category theory can provide help with at least the following:

- *Formulating definitions and theories*. In computing science, it is often more difficult to formulate concepts and results than to give a proof. The seven guidelines of this paper can help with formulation; the guidelines can also be used to measure the elegance and coherence of existing formulations.
- *Carrying out proofs*. Once basic concepts have been correctly formulated in a categorical language, it often seems that proofs "just happen": at each step, there is a "natural" thing to try, and it works. Diagram chasing (see Section 1.2) provides many examples of this. It could almost be said that the purpose of category theory is to reduce all proofs to such simple calculations.
- *Discovering and exploiting relations with other fields*. Sufficiently abstract formulations can reveal surprising connections. For example, an analogy between Petri nets and the λ-calculus might suggest looking for a closed category structure on the category of Petri nets [52].
- *Dealing with abstraction and representation independence*. In computing science, more abstract viewpoints are often more useful, because of the need to achieve independence from the often overwhelmingly complex details of how things are represented or implemented. A corollary of the first guideline (given

[1]The research reported in this paper was supported in part by grants from the UK Science and Engineering Research Council, the National Science Foundation, and the System Development Foundation, as well as contracts with the Office of Naval Research and the Fujitsu Corporation.

in Section 1) is that two objects are "abstractly the same" iff they are isomorphic; see Section 1.1. Moreover, universal constructions (i.e., adjoints) define their results uniquely up to isomorphism, i.e., "abstractly" in just this sense.

- *Formulating conjectures and research directions.* Connections with other fields can suggest new questions in your own field. Also the seven guidelines can help to guide research. For example, if you have found an interesting functor, then you might be well advised to investigate its adjoints.

- *Unification.* Computing science is very fragmented, with many differenct subdisciplines having many different schools within them. Hence, we badly need the kind of conceptual unification that category theory can provide.

Category theory can also be abused, and in several different styles. One style of abuse is *specious generality*, in which some theory or example is generalised in a way that does not actually include any new examples of genuine interest. A related style of abuse is *categorical overkill*, in which the language of category theory is used to describe phenomena that do not actually require such an elaborate treatment or terminology. An example is to describe a Galois connection in the language of adjoint functors.

Category theory has been called "abstract nonsense" by both its friends and its detractors. Perhaps what this phrase suggests to both is that category theory has relatively more form than content, compared to other areas of mathematics. Its friends claim this as a virtue, in contrast to the excessive concreteness and representation dependence of set theoretic foundations, and the relatively poor guidance for discovering elegant and coherent theories that they provide. Section 9 discusses this further.

Category theory can also be used in quite concrete ways, because categories are after all just another algebraic structure, generalising both monoids and partial orders (see also Example 1.4 below).

This paper presents seven guidelines for using category theory, each with some general discussion and specific examples. There is no claim to originality, because I believe the underlying intuitions are shared by essentially all workers in category theory, although they have been perhaps understandably reluctant to place such dogmatic assertions in textbooks or other written documents[2]. The guidelines are necessarily imprecise, and will seem exaggerated if taken too literally, because they are not objective facts, but rather heuristics for applying certain mathematical concepts. In particular, they may seem difficult to apply, or even impossible, in some situations, and they may need refinement in others. As a reminder that they should not be taken too dogmatically, I will call them *dogmas*.

No attempt is made to be exhaustive. In particular, the technical definitions are omitted, because our purpose is to provide intuition, and the definitions can be found in any textbook. Thus, if you are a newcomer to category theory, you will need to use some text in connection with this paper. Unfortunately, no existing text is ideal for computing scientists, but perhaps that by Goldblatt [36] comes closest. The classic text by Mac Lane [47] is warmly recommended for those with sufficient mathematics background, and Herrlich and Strecker's book [39] is admirably thorough; see also

[2]As far as I know, the first such attempt is my own in [34], which gives four of the guidelines here. The only other attempt that I know is due to Lambek and Scott [45], who give a number of "slogans" in a similar style.

[2] and [45]. The paper [22] gives a relatively concrete and self-contained account of some basic category theory for computing scientists, using theories, equations, and unification as motivation, and many examples from that paper are used here.

1 Categories

The first dogma is as follows:

> *To each species of mathematical structure, there corresponds a* **category** *whose objects have that structure, and whose morphisms preserve it.*

It is part of this guideline that in order to understand a structure, it is necessary to understand the morphisms that preserve it. Indeed, category theorists have argued that morphisms are more important than objects, because they reveal what the structure really is. Moreover, the category concept can be defined using only morphisms. Perhaps the bias of modern Western languages and cultures towards objects rather than relationships accounts for this (see [50, 64] for some related discussion). By way of notation, we use ";" for composition, and 1_A for the identity morphism at an object A. Now some examples:

1.1 **Sets.** If we take sets to be objects, then their morphisms are clearly going to be functions. A set morphism, however, is not just a set of ordered pairs, because it must also specify particular source and target sets. This is consistent with practice in computation theory which assigns types to functions. The set theoretic representation of functions is an artifact of the set theoretic foundations of mathematics, and like all such representations, has accidental properties beyond those of the concept it is intended to capture. One of those properties is that any two sets of ordered pairs can be composed to yield a third. The category $\mathcal{S}et$ of sets embodies a contrary point of view, that each function has a domain in which its arguments are meaningful, a codomain in which its results are meaningful, and composition of functions is only allowed when meaningful in this sense. (See [36] for related discussions.)

1.2 **Relations.** Just as with functions, it seems desirable to take the view that the composition of relations is only meaningful when the domains match. Thus, we may define a **relation** from a set A_0 to a set A_1 to be a triple (A_0, R, A_1) with $R \subseteq A_0 \times A_1$, and then allow its composition with (B_0, S, B_1) to be defined iff $A_1 = B_0$. This gives rise to a category that we denote $\mathcal{R}el$, of which $\mathcal{S}et$ can be considered a subcategory.

1.3 **Graphs.** A **graph** G consists of a set E of **edges**, a set N of **nodes**, and two functions $\partial_0, \partial_1 \colon E \to N$ which give the **source** and **target** of each edge, respectively. Because the major components of graphs are sets, the major components of their morphisms should be corresponding functions that preserve the additional structure. Thus a morphism from $G = (E, N, \partial_0, \partial_1)$ to $G' = (E', N', \partial_0', \partial_1')$ consists of two functions, $f \colon E \to E'$ and $g \colon N \to N'$, such that the following diagram commutes in $\mathcal{S}et$ for $i = 0, 1$:

$$\begin{tikzcd} E \arrow[r, "\partial_i"] \arrow[d, "f"'] & N \arrow[d, "g"] \\ E' \arrow[r, "\partial'_i"'] & N' \end{tikzcd}$$

To show that we have a category $\mathcal{G}raph$ of graphs, we must show that a composition of two such morphisms is another, and that a pair of identity functions satisfies the diagrams and also serves as an identity for composition.

1.4 **Paths in a Graph.** Given a graph G, each path in G has a source and a target node in G, and two paths, p and p', can be composed to form another path $p;p'$ iff the source of p' equals the target of p. Clearly this composition is associative when defined, and each node can be given an "identity path" having no edges. This category is denoted $\mathcal{P}a(G)$. (Details may be found in [47], [34], [19], and many other places.)

1.5 **Automata.** An automaton consists of an input set X, a state set S, an output set Y, a transition function $f\colon X \times S \to S$, an initial state $s_0 \in S$, and an output function $g\colon S \to Y$. What does it mean to preserve all this structure? Because the major components of automata are sets, the major components of their morphisms should be corresponding functions that preserve the structure. Thus a morphism from $A = (X, S, Y, s_0, f, g)$ to $A' = (X', S', Y', s'_0, f', g')$ should consist of three functions, $h\colon X \to X'$, $i\colon S \to S'$, and $j\colon Y \to Y'$, such that the following diagrams commute in $\mathcal{S}et$:

$$\begin{tikzcd} & S \arrow[dd, "i"] \\ \{*\} \arrow[ur, "s_0"] \arrow[dr, "s'_0"'] & \\ & S' \end{tikzcd} \qquad \begin{tikzcd} X\times S \arrow[r, "f"] \arrow[d, "h\times i"'] & S \arrow[d, "i"] \\ X'\times S' \arrow[r, "f'"'] & S' \end{tikzcd} \qquad \begin{tikzcd} S \arrow[r, "g"] \arrow[d, "i"'] & Y \arrow[d, "j"] \\ S' \arrow[r, "g'"'] & Y' \end{tikzcd}$$

where $\{*\}$ denotes an arbitrary one point set (with point $*$). It must be shown that a composition of two such morphisms is another, and that a triple of identities satisfies the diagrams and serves as an identity for composition. These checks show that we have a category $\mathcal{A}ut$ of automata, and their simplicity increases our confidence in the correctness of the definitions [18].

1.6 **Types.** Types are used to classify "things," and according to the first dogma, they should form a category having types as objects; of course, depending on what is being classified, different categories will arise.

A simple example is finite product types, which are conveniently represented by natural numbers, with morphisms that describe what might be called "register transfer operations" among tuples of "registers". Thus, $n \in \omega$ indicates an n-tuple $\langle t_1, ..., t_n\rangle$ of data items in n "registers," and a morphism $f: m \to n$ is a function $\{1, ..., n\} \to \{1, ..., m\}$ indicating that the content of register $f(i)$ should be transferred to register i, for $i = 1, ..., n$. In fact, if we identify the number n with the set $\{1, ..., n\}$ (and 0 with $\emptyset$), then this category is the *opposite* of a subcategory of $\mathcal{S}et$; let us denote it $\mathcal{N}$. A variant of $\mathcal{N}$ has as its objects the finite subsets of a fixed countable set X, and as morphisms again the opposites of functions among these, so that we get another opposite of a subcategory of $\mathcal{S}et$, denoted say $\mathcal{X}$. Here, the "registers" are denoted by "variable symbols" from X, rather than by natural numbers. Going a little further, we can assign sorts from a set S to the symbols in X, and require that the morphisms preserve these sorts. Let us denote this category $\mathcal{X}_S$.

1.7 **Substitutions.** Two key attributes of a substitution are the set of variables into which it substitutes, and the set of variables that occur in what it substitutes. Thus, substitutions have natural source and target objects, each a set of variables, as in Example 1.6 above. Clearly there are identity substitutions for each set of variables (substituting each variable for itself), and the composition of substitutions is associative when defined. Thus, we get a category with substitutions as morphisms.

1.8 **Theories.** In his 1963 thesis [49], F.W. Lawvere developed a very elegant approach to universal algebra, in which an *algebraic theory* is defined to be a category $\mathbb{T}$ whose morphisms correspond to equivalence classes of terms, and whose objects indicate the variables involved in these terms, as in Example 1.6 above. In this approach, theories are closed under finite products (as defined in Example 4.1 below). Although Lawvere's original development was unsorted, it easily extends to the many-sorted case, and in many other ways, including the so-called "sketches" studied by Ehresmann, Gray, Barr, Wells, and others; for example, see [3]. Of course, all the theories of a given kind form a category.

1.1 Isomorphism

One very simple, but still significant, fruit of category theory is a general definition of isomorphism, suitable for any species of structure at all: a morphism $f\colon A \to B$ is an **isomorphism** in a category $\mathcal{C}$ iff there is another morphism $g\colon B \to A$ in $\mathcal{C}$ such that $g;f = 1_A$ and $f;g = 1_B$. In this case, the objects A and B are **isomorphic**. It is a well established principle in abstract algebra, and now in other fields as well, that isomorphic objects are abstractly the same, or more precisely:

> *Two objects have the same structure iff they are* **isomorphic**, *and an "abstract object" is an isomorphism class of objects.*

This demi-dogma can be seen as a corollary of the first dogma. It provides an immediate check on whether or not some structure has been correctly formalised: unless it is satisfied, the objects, or the morphisms, or both, are wrong. This principle is so pervasive that isomorphic objects are often considered the same, and "the X" is used instead of "an X" when X is actually only defined up to isomorphism. In computing science, this principle guided the successful search for the right definition of "abstract data type" in [33].

1.2 Diagram Chasing

A useful way to get an overview of a problem, theorem, or proof, is to draw one or more diagrams that show the main objects and morphisms involved. A diagram

commutes iff whenever p and p' are paths with the same source and target, then the compositions of the morphisms along these two paths are equal. The fact that pasting two commutative diagrams together along a common edge yields another commutative diagram provides a basis for a purely diagrammatic style of reasoning about equality of compositions. Because it is valid for diagrams in any category whatever, this proof style is very widely applicable; for example, it applies to substitutions (as in Example 1.5). Moreover, it has been extended with conventions for pushouts, for uniqueness of morphisms, and for certain other common situations. Often proofs are suggested just by drawing diagrams for what is known and what is to be proved. A simple illustration from Example 1.3 is to prove that a composition of two graph morphisms is another graph morphism; all we have to do is paste together the corresponding diagrams for the two morphisms.

2 Functors

The second dogma says:

> *To any natural construction on structures of one species, yielding structures of another species, there corresponds a* **functor** *from the category of the first species to the category of the second.*

It is part of this dogma that a construction is not merely a function from objects of one species to objects of another species, but must also preserve the essential relationships among objects, including their structure preserving morphisms, and compositions and identities for these morphisms. This provides a test for whether or not the construction has been properly formalised. Of course, functoriality does not *guarantee* correct formulation, but it can be surprisingly helpful in practice. Now some examples:

2.1 **Free Monoids.** It is quite common in computing science to construct the free monoid X^* over a set X. It consists of all finite strings $x_1 \ldots x_n$ from X, including the empty string Λ. This construction gives a functor from the category of sets to the category of monoids, with a function $f\colon X \to Y$ inducing $f^*\colon X^* \to Y^*$ by sending Λ to Λ, and sending $x_1 \ldots x_n$ to $f(x_1) \ldots f(x_n)$. This functor is called the "polymorphic list type constructor" in functional programming.

2.2 **Behaviours.** Given an automaton $A = (X, S, Y, f, g)$, its *behaviour* is a function $b : X^* \to Y$, from the monoid X^* of all strings over X, to Y, defined by $b(u) = g(\overline{f}(u))$, where $\overline{f}$ is defined by $\overline{f}(\Lambda) = s_0$ and $\overline{f}(ux) = f(x, \overline{f}(u))$, for $x \in X$ and $u \in X^*$. This construction should be functorial. For this, we need a category of behaviours. The obvious choice is to let its objects be pairs $(X, b\colon X^* \to Y)$ and to let its morphisms from $(X, b\colon X^* \to Y)$ to $(X', b'\colon X'^* \to Y')$ be pairs (h, j) where $h\colon X \to X'$ and $j\colon Y \to Y'$, such that the diagram

$$\begin{tikzcd} X^* \arrow[r, "b"] \arrow[d, "h^*"'] & Y \arrow[d, "j"] \\ X'^* \arrow[r, "b'"'] & Y' \end{tikzcd}$$

commutes in $\mathcal{S}et$. Denote this category $\mathcal{B}eh$ and define $B\colon \mathcal{A}ut \to \mathcal{B}eh$ by $B(X,S,Y,f,g) = g;\overline{f}$ and $B(h,i,j) = (h,j)$. That this *is* a functor helps to confirm the elegance and coherence of the previous definitions. See [18].

2.3 **Models.** In the Lawvere approach to universal algebra [49], an algebra is a functor from a theory $\mathbb{T}$ to $\mathcal{S}et$. Here, "construction" takes the meaning of "interpretation": the abstract structure in $\mathbb{T}$ is interpreted (i.e., constructed) concretely in $\mathcal{S}et$, i.e., these functors must preserve finite products. More generally, if $\mathbb{T}$ is some kind of theory, then "models" of $\mathbb{T}$ are functors $M : \mathbb{T} \to \mathcal{S}et$ that preserve the structure of these theories, e.g., finite products. More generally, we can take models of $\mathbb{T}$ in a suitable category $\mathcal{C}$ with finite products, as finite product preserving functors. For example, many sorted algebras arise as functors from a theory over the type system $\mathcal{X}_S$; Example 1.5 can be seen as an example of this, by taking S to have three elements.

2.4 **Forget It.** If all widgets are whatsits, then there is a "forgetful functor" from the category of widgets to the category of whatsits. For example, every group is a monoid by forgetting its inverse operation, and every monoid is a semigroup by forgetting its identity. Notice that a ring (with identity) is a monoid in *two different* ways, one for its additive structure and one for its multiplicative structure.

2.5 **Categories.** Of course, the (small) categories also form a category, with functors as morphisms. It is denoted $\mathcal{C}at$.

2.6 **Diagrams and the Path Category Construction.** The construction in Example 1.4 of the category $\mathcal{P}a(G)$ of all paths in a graph G gives rise to a functor $\mathcal{P}a\colon \mathcal{G}raph \to \mathcal{C}at$ from graphs to categories. Then a **diagram** in a category $\mathcal{C}$, with **shape** a graph G, is a functor D: $\mathcal{P}a(G) \to \mathcal{C}$. It is conventional to write just $D : G \to \mathcal{C}$, and even to call D a "functor," because D: $\mathcal{P}a(G) \to \mathcal{C}$ is in fact fully determined by its restriction to G, which is a graph morphism; see Example 6.2 below.

2.7 **Programs and Program Schemes.** A non-deterministic flow diagram **program** P with parallel assignments, go-to's, and arbitrary built-in data structures, including arbitrary functions and tests, can be seen as a functor from a graph G (the program's "shape") into the category $\mathcal{R}el$ whose objects are sets and whose morphisms are relations. An edge $e\colon n \to n'$ in G corresponds to a program statement, and the relation $P(e) : P(n) \to P(n')$ gives its semantics. For example, the test "if $X > 2$" on natural numbers corresponds to the partial identity function $\omega \to \omega$ defined iff $X > 2$, and the assignment "$X := X - 1$" corresponds to the partial function $\omega \to \omega$ sending X to $X - 1$ when $X > 0$. The **semantics** of P with input node n and output node n' is then given by the formula

$$P(n,n') = \bigcup\{P(p) \mid p\colon n \to n' \in \mathcal{P}a(G)\}.$$

This approach originated in Burstall [5]. Techniques that allow programs to have *syntax* as well as semantics are described in [19][3]: A **program scheme** is a functor $P\colon G \to \mathbb{T}$ into a theory $\mathbb{T}$ "enriched" with a partial order structure on its morphism sets $\mathbb{T}(A,B)$ (the reader familiar with 2-categories should note that this makes $\mathbb{T}$ a 2-category). A semantics for statements then arises by giving a functor $A\colon \mathbb{T} \to \mathcal{R}el$, that is, an interpretation for $\mathbb{T}$, also called a $\mathbb{T}$-algebra. The semantics of a program is then computed by the above formula for the composition $P;A\colon G \to \mathcal{R}el$. There seems to be much more research that could be done in this area. For example, [29] gives an inductive proof principle for collections of mutually recursive procedures, and it would be interesting to consider other program constructions in a similar setting.

2.8 **Theory Interpretations.** Extending the discussion in Example 2.3, an "interpretation" of a theory T in a theory T' should be a functor $F\colon T \to T'$ which preserves theory structure (e.g., types and finite products). Such functors are the *same thing* as **theory morphisms**. In particular, interpretations of program schemes, which of course are programs, will arise in this way.

2.9 **Polymorphic Type Constructors.** If we think of the types of a functional programming language as forming a category $\mathcal{T}$, with objects like `Int` and `Bool`, then polymorphic type constructors, like `list`, are endofunctors on $\mathcal{T}$, that is, functors $\mathcal{T} \to \mathcal{T}$; some others would be `set` and `list` $\times$ `list`, the latter sending a type α to the type $\mathtt{list}(\alpha) \times \mathtt{list}(\alpha)$.

3 Naturality

The third dogma says:

> *To each natural translation from a construction $F\colon \mathcal{A} \to \mathcal{B}$ to a construction $G\colon \mathcal{A} \to \mathcal{B}$ there corresponds a* **natural transformation** *$F \Rightarrow G$.*

Although this looks like a mere definition of the phrase "natural translation," it can nevertheless be very useful in practice. It is also interesting that this concept was the historical origin of category theory, since Eilenberg and Mac Lane [11] used it to formalise the notion of an equivalence of homology theories, and then found that for this definition to make sense, they had to define functors, and for functors to make sense, they had to define categories. (This history also explains why homology theory so often appears in categorical texts, and hence why so many of them are ill-suited for computing scientists.) Now some examples:

3.1 **Homomorphisms.** As already indicated, in the Lawvere approach to universal algebra, algebras are functors, and so we should expect homomorphisms to be natural transformations; and indeed, they are.

3.2 **Natural Equivalence.** A natural transformation $\eta\colon F \Rightarrow G$ is a natural equivalence iff each $\eta_A\colon F(A) \to G(A)$ is an isomorphism. This is the natural notion of isomorphism for functors, and is equivalent to the existence of

[3] Only the original 1972 conference version contains this definition.

$\nu\colon G \Rightarrow F$ such that $\nu;\eta = 1_F$ and $\eta;\nu = 1_G$. This is also exactly the concept that motivated Eilenberg and Mac Lane, and in the context of Example 3.1, it specialises to isomorphism of algebras.

3.3 **Data Refinement.** A graph with its nodes labelled by types and its edges labelled by function symbols can be seen as an impoverished Lawvere theory that has no equations and no function symbols with more than one argument. However, such theories still admit algebras, which are functors into $\mathcal{S}et$, and homomorphisms, which of course are natural transformations. These algebras can be viewed as *data representations* for the basic data types and functions of a programming language, and their homomorphisms can be viewed as *data refinements*. Considered in connection with the basic program construction operations of a language, this can lead to some general techniques for developing correct programs [40]. It would be interesting to extend this to more general variants of Lawvere theories (such as many-sorted theories or sketches), and to the more general data representations studied in the abstract data type literature (e.g., [33, 9]).

3.4 **Program Homomorphisms.** Because Example 2.7 defines programs as functors, we expect program homomorphisms to be natural transformations between programs. Indeed, Burstall [5] shows that a weak form of Milner's program simulations [53] arises in just this way. In [19], this is generalised to programs that may have different shapes, and to maps from edges to paths, by defining a **homomorphism** from $P_0 : G_0 \to \mathcal{C}$ to $P_1 : G_1 \to \mathcal{C}$ to consist of a functor $F\colon G_0 \to \mathcal{P}a(G_1)$ and a natural transformation $\eta\colon P_0 \to F;P_1$. Some theory and applications for this are also given in [19], including techniques for proving correctness, termination, and equivalence, by unfolding programs into equivalent infinite trees.

3.5 **Polymorphic Functions.** If polymorphic type constructors are functors (as in Example 2.8), then polymorphic functions should be natural transformations; and indeed, they are. Examples include

```
append :  list list -> list
```

and

```
reverse :  list -> list
```

3.6 **Functor Categories.** Let $\mathcal{A}$ and $\mathcal{B}$ be categories. Then there is a category, denoted $\mathcal{C}at[\mathcal{A},\mathcal{B}]$, whose objects are the functors from $\mathcal{A}$ to $\mathcal{B}$, and whose morphisms are natural transformations. In particular, if $\mathbb{T}$ is a theory, then the $\mathbb{T}$-algebras are a subcategory of $\mathcal{C}at[\mathbb{T},\mathcal{S}et]$.

4 Limits

The fourth dogma says:

> *A* **diagram** *D in a category $\mathcal{C}$ can be seen as a system of constraints, and then a* **limit** *of D represents all possible solutions of the system.*

In particular, if the diagram represents some physical (or conceptual) system, then the limit provides an object which (together with its projection morphisms) represents all possible behaviours of the system that are consistent with the given constraints. This intuition goes back to some work on General System Theory from 1969-74, [16, 27], and has many applications in computing science:

4.1 **Products.** An early achievement of category theory was to give a precise definition for the notion of "product," which was previously known in many special cases, but only understood vaguely as a general concept. The definition is due to Mac Lane [46].

4.2 **Product Types.** Given types T_1 and T_2, their "parallel composition" is their product in the category $\mathcal{T}$ of types. Thus, a morphism $f : T_1 \times T_2 \to T$ takes two "inputs" in parallel, of types T_1 and T_2, and returns one output, of type T. It is usual to assume that a category of types used in defining some kind of theory has finite products, including an empty product (the product of no factors, i.e., a final object), usually denoted 1. Both $\mathcal{N}$ and $\mathcal{X}$ are subcategories of $\mathcal{S}et^{op}$, and products in them are disjoint unions in $\mathcal{S}et$.

4.3 **Theories.** A **generalised Lawvere theory** $\mathbb{T} : \mathcal{T} \to \mathcal{A}$ over a type system $\mathcal{T}$ (assumed to have finite products) is a finite product preserving functor that is surjective on objects, from $\mathcal{T}$ to a category $\mathcal{A}$ with finite products. Except for "degenerate" cases, a theory $\mathbb{T} : \mathcal{T} \to \mathcal{A}$ is bijective on objects, and we can assume that $|\mathcal{T}| = |\mathcal{A}|$ and that $\mathcal{T}$ is a subcategory of $\mathcal{A}$; hence, we may identify $\mathbb{T}$ and $\mathcal{A}$.

A **morphism** of theories over $\mathcal{T}$ is a finite product preserving functor which also preserves $\mathcal{T}$. An **algebra** of a theory $\mathbb{T} : \mathcal{T} \to \mathcal{A}$ is a finite product preserving functor to $\mathcal{S}et$ (or more generally, to a category $\mathcal{C}$ with finite products). Of course, **homomorphisms** of $\mathbb{T}$-algebras are natural transformations, giving a category of $\mathbb{T}$-algebras. When $\mathcal{T} = \mathcal{N}$, we get the classical unsorted general algebras, in Lawvere form. When $\mathcal{T} = \mathcal{X}$, with $\mathcal{X}$ S-sorted, we get S-sorted general algebras. [22] also discusses congruences and quotients of generalised Lawvere theories.

4.4 **Equations and unification.** We can think of a pair $f, g : T \to T'$ of morphisms in a theory as an **equation**. Then, by the fourth dogma, the most general solution of this equation is given by the *equaliser* of f and g, if it exists. For the classical case of unsorted, anarchic (i.e., obeying no laws) theories, the morphisms are terms, and equalisers give *most general unifiers*. More general kinds of unification arise by going to more general kinds of theories; for example, imposing associative and commutative laws on some operations in the theory leads to so-called AC-unification. For some theories, only **weak equalisers** can be found; these weaken the "there exists a unique morphism" requirement to mere existence. In fact, weak equalisers formalise the classical definition of unifiers; nonetheless, the stronger condition is often satisfied in practice. Generalising again, a **system of constraints** is a diagram in a theory, and its **most general solution** is given by its limit, if it exists.

There are many examples of this situation: solving systems of linear equations; polymorphic type inference; unification in the sense of "unification grammars" in linguistics; solving Scott domain equations; and least fixpoints. All these examples (and some others) are discussed in more detail in [22], as are some techniques for proving that unifiers exist. Another example is the justification of the formula in Example 2.7 for the semantics of a program.

5 Adjoints

The fifth dogma says:

> *To any canonical construction from one species of structure to another corresponds an* **adjunction** *between the corresponding categories.*

Although this can be seen as just a definition of "canonical construction," it can be very useful in practice. The essence of an adjoint is the *universal property* that is satisfied by its value objects. This property says that there is a unique morphism satisfying certain conditions. It is worth noting that any two (right, or left) adjoints to a given functor are naturally equivalent, i.e., adjointness determines a construction uniquely up to isomorphism. Now some examples:

5.1 **Products and Sums.** Many of the constructions described above are intuitively canonical, and hence are adjoints. For example, binary products in a category $\mathcal{C}$ give a functor $\Pi\colon \mathcal{C}\times\mathcal{C} \to \mathcal{C}$, which is left adjoint to $\Delta : \mathcal{C} \to \mathcal{C}\times\mathcal{C}$, the "diagonal" functor, sending an object C in $\mathcal{C}$ to the pair (C, C), and sending a morphism $c\colon C \to C'$ in $\mathcal{C}$ to $(c, c) : (C, C) \to (C', C')$ in $\mathcal{C}\times\mathcal{C}$. Moreover, $\mathcal{C}$ has coproducts (also called "sums") iff Δ has a right adjoint. This beautifully simple way to formalise two mathematical concepts of basic importance is due to Mac Lane [46], and extends to general limits and colimits.

5.2 **Freebies.** Another beautifully simple formalisation gives a general definition of "free" constructions: they are the left adjoints of forgetful functors. For example, the path category functor $\mathcal{P}a\colon \mathcal{G}raph \to \mathcal{C}at$ of Example 2.6 is left adjoint to the forgetful functor $\mathcal{C}at \to \mathcal{G}raph$, and thus may be said to give the free category over a graph.

5.3 **Minimal Realisation.** An automaton (X, S, Y, f, g) is **reachable** iff its function $\overline{f}\colon X^* \to S$ is surjective. Let $\mathcal{A}$ denote the subcategory of $\mathcal{A}ut$ whose objects are reachable and whose morphisms (i, j, k) have i surjective. Then the restriction $B\colon \mathcal{A} \to \mathcal{B}eh$ of $B : \mathcal{A}ut \to \mathcal{B}eh$ to $\mathcal{A}$ has a right adjoint which gives the minimal realisation of a behaviour [18]. Because right adjoints are uniquely determined, this provides a convenient abstract characterisation of minimal realisation. Moreover, this characterisation extends to, and even suggests, more general minimal realisation situations, e.g., see [17].

5.4 **Syntax and Semantics.** One of the more spectacular adjoints is that between syntax and semantics for algebraic theories, again due to Lawvere in his thesis; see [49].

5.5 **Cartesian Closed Categories.** A Cartesian closed category has binary products, and a right adjoint to each functor sending A to $A\times B$. It is remarkable that this concept turns out to be essentially the (typed) λ-calculus; see [45]. This connection has been used, for example, as a basis for the efficient compilation of higher order functional languages [8]. An advantage is that optimisation techniques can be proved correct by using purely equational reasoning.

5.4 **Kleisli Categories.** Another way to generalise Lawvere theories is to view an arbitrary adjunction as a kind of theory. So-called **monads** (also called **triples**) are an abstraction of the necessary structure, and the **Kleisli category** over a monad gives the category of free algebras [47]. Again, there are surprisingly many examples. The paper [22] shows how a Kleisli category generates a generalised Lawvere theory, and then shows that many different problems of unification (that is, of solving systems of equations) can be naturally formulated as finding equalisers in Kleisli categories. Examples include unification in order sorted and continuous theories. Moggi [55] uses Kleisli categories to get an abstract notion of "computation" which gives rise to many interesting generalisations of the λ-calculus.

6 Colimits

The sixth dogma says:

> *Given a species of structure, say widgets, then the result of interconnecting a system of widgets to form a super-widget corresponds to taking the* **colimit** *of the diagram of widgets in which the morphisms show how they are interconnected.*

At least for me, this intuition arose in the context of General Systems Theory [16, 27]. It may be interesting to note that the duality between the categorical definitions of limits and colimits suggests a similar duality between the intuitive notions of solution and interconnection. Now some examples:

6.1 **Putting Theories together to make Specifications.** Complexity is a fundamental problem in programming methodology: large programs, and their large specifications, are very difficult to produce, to understand, to get right, and to modify. A basic strategy for defeating complexity is to break large systems into smaller pieces that can be understood separately, and that when put back together give the original system. If successful, this in effect "takes the logarithm" of the complexity. In the semantics of Clear [6, 7], specifications are represented by theories, in essentially the same sense as Lawvere (but many-sorted, and with signatures), and specifications are put together by colimits in the category of such theories. More specifically, the application of a generic theory to an actual is computed by a pushout. OBJ [13, 28, 14], Eqlog [30], and FOOPS [31] extend this notion of generic module to functional, logic (i.e., relational), and object oriented programming, and in their combinations. It has even been applied to Ada [21, 63].

6.2 **Graph Rewriting.** Another important problem in computing science is to find *models of computation* that are suitable for massively parallel machines. A successful model should be abstract enough to avoid the implementation details of particular machines, and yet concrete enough to serve as an intermediate target language for compilers. Graph rewriting provides one promising area within which to search for such models [43, 32, 15, 41], and colimits seem to be quite useful here [10, 58, 44]. Graph rewriting is also important for the unification grammars that are now popular in linguistics [60, 22]. There seem to be many opportunities for further research in these areas.

6.3 **Initiality.** The simplest possible diagram is the empty diagram. Its colimit is an *initial object*, which is more simply explained as an object that has a unique morphism to any object. Like any adjoint, it is determined uniquely up to isomorphism, so any two initial objects in a category are isomorphic (of course, this can also be shown directly); hence, initiality gives a convenient way to define entities "abstractly". It is also worth mentioning that universality can be reduced to initiality (in a comma category), and hence so can colimits.

6.4 **Initial Model Semantics.** It seems remarkable that initiality is so very useful in computing science. Beginning with the formalisation of abstract syntax as an initial algebra [20], initiality has been applied to an increasing range of fundamental concepts, including induction and recursion [35, 51], abstract data types [33], domain equations (see below), computability [51], and model theoretic semantics for functional [13], logic (i.e., relational), combined functional and relational, and constraint logic [30] programming languages. The latter is interesting because it involves initiality in a category of model extensions, i.e., of morphisms, rather than just models. In general, this research can be seen as formalising, generalising, and smoothing out the classical Herbrand Universe construction [38], and it seems likely that much more interesting work can be done along these lines.

6.5 **Solving Domain Equations.** Scott [59] presents an "inverse limit" construction for solving domain equations, and records some suggestions by Lawvere that clarify this construction by viewing it as a colimit in an associated category of retracts. These ideas are taken further in [61], which also generalises from partial orders to categories and shows that least fixpoints are initial algebras, among other things. A key construction is the colimit of an infinite sequence of morphisms, generalising the traditional construction $\bigsqcup_{n\in\omega} F^n(\perp)$ of a least fixpoint.

7 Comma Categories

The seventh dogma says:

> *Given a species of structure $\mathcal{C}$, then a species of structure obtained by "decorating" or "enriching" that of $\mathcal{C}$ corresponds to a* **comma category** *under $\mathcal{C}$ (or under a functor from $\mathcal{C}$).*

It seems more difficult to be precise about this intuition than the others, but hopefully some examples will help to clarify things. The following are just a few of the many examples that can be found in computing science:

7.1 **Graphs.** Many categories of graph are comma categories. For example, if $2\times$ denotes the functor $\mathcal{S}et \to \mathcal{S}et$ sending S to $S \times S$, then the category $\mathcal{G}raph$ of Example 1.3 is the comma category $(\mathcal{S}et \downarrow 2\times)$.

7.2 **Labelled Graphs.** Given some category $\mathcal{G}$ of graphs and a forgetful functor $\mathcal{U}\colon \mathcal{G} \to \mathcal{S}et$, say giving the node set of graphs in $\mathcal{G}$, and given a set L to be used for node labels, then the comma category $(\mathcal{U} \downarrow L)$ is the category of graphs from $\mathcal{G}$ with nodes labelled by L. In the same way, we can decorate edges of graphs, or branches of trees.

7.3 **Theories.** If $\mathcal{FPC}at$ is the category of categories with finite products, with finite product preserving functors as morphisms, and if $\mathcal{T}$ is a type system (i.e., an object in $\mathcal{FPC}at$), then the category of theories over $\mathcal{T}$ is $(\mathcal{T} \downarrow \mathcal{FPC}at)$.

Comma categories are another basic construction that first appeared in Lawvere's thesis. They tend to arise when morphisms are used as objects. Viewing a category as a comma category also makes available some general results to prove the existence of limits and colimits [25].

8 Further Topics

Although they are particularly fundamental, the seven dogmas given above far from exhaust the richness of category theory. This section mentions some further categorical constructions, about each of which one might express surprise at how many examples there are in computing science.

8.1 **2-Categories.** Sometimes morphisms not only have their usual composition, identity, source and target, but *also* serve as objects for some other, higher-level, morphisms. This leads to 2-categories, of which the category $\mathcal{C}at$ of categories is the canonical example, with natural transformations as morphisms of its morphisms. This concept was mentioned in Example 2.7, and is also used in [24], [26], [40], [56], among other places, and is mentioned in [61].

8.2 **Monoidal Categories.** There are many cases where a category has a natural notion of multiplication that is not the usual Cartesian product but nevertheless enjoys many of the same properties. The category of Petri nets studied in [52] has already been mentioned, and a variety of recent work suggests that monoidal categories may be broadly useful in understanding the relationships among the various theories of concurrency, e.g., see [12].

8.3 **Indexed Categories.** A strict indexed category is just a functor $\mathcal{B}^{op} \to \mathcal{C}at$. The papers [62] and [23] give many examples of indexed categories in computing science, and [62] gives some general theorems, including simple sufficient conditions for completeness of the associated "Grothendieck" category. Moggi [56] applies indexed categories to programming languages, and in particular shows how to get a kind of higher order module facility for languages like ML. (Non-strict indexed categories are significantly more complex, and have been used in foundational studies [57].)

8.5 **Topoi.** A profound generalisation of the idea that a theory is a category appears in the *topos* notion developed by Lawvere, Tierney, and others. In a sense, this notion captures the essence of set theory. It also has surprising relationships to algebraic geometry, computing science, and intuitionistic logic [36, 2, 42].

9 Discussion

The traditional view of foundations requires giving a system of axioms, preferably first order, that assert the existence of certain primitive objects with certain properties, and of certain primitive constructions on objects, such that all objects of interest can be constructed, and all their relevant properties derived, within the system. The axioms should be as self-evident, as few in number, and as simple, as possible, in order to nurture belief in their consistency, and to make them as easy to use as possible. This approach is inspired by the classical Greek account of plane geometry.

The best known foundation for mathematics is set theory, which has been very successful at constructing the objects of greatest interest in mathematics. It has, however, failed to provide a commonly agreed upon set of simple, self-evident axioms. For example, classical formulations of set theory (such as Zermello-Frankel) have been under vigorous attack by intuitionists for nearly eighty years. More recently, there has been debate about whether the Generalised Continuum Hypothesis is "true," following the originally startling proof (by Paul Cohen) that it is independent of other, more widely accepted axioms of set theory. Still more recently, there has been debate about the Axiom of Foundation, which asserts that there is no infinite sequence of sets $S_1, S_2, S_3, ...$ such that each S_{i+1} is an element of S_i. In fact, Aczel [1] and others have used an *Anti*-Foundation Axiom, which positively asserts the existence of such non-well founded sets, to model various phenomena in computation, including communicating processes in the sense of Milner [54]. I think it is fair to say that most mathematicians no longer believe in the heroic ideal of a single generally accepted foundation for mathematics, and that many no longer believe in the possibility of finding "unshakable certainties" [4] upon which to found all of mathematics.

Set theoretic foundations have also failed to provide fully satisfying accounts of mathematical practice in certain areas, including category theory itself, and moreover have encouraged research into areas that have little or nothing to do with mathematical practice, such as large cardinals. (Mac Lane [48] gives a lively discussion of these issues; see also [37] for an overview of various approaches to foundations.) In any case, attempts to find a minimal set of least debatable concepts upon which to erect mathematics have little direct relevance to computing science. Of course, the issue no longer seems as urgent as it once did, because no new paradoxes have been discovered for a long time.

This paper has tried to show that category theory provides a number of broadly useful, and yet surprisingly specific, guidelines for organising, generalising, and discovering analogies among and within various branches of mathematics and its applications. I wish to suggest that the existence of such guidelines can be seen to support an alternative, more pragmatic view:

> *Foundations should provide general concepts and tools that reveal the structures and interrelations of various areas of mathematics and its applications, and that help in doing and using mathematics.*

In a field which is not yet very well developed, such as computing science, where it often seems that getting the definitions right is the hardest task, foundations in this sense can be very useful, because they can suggest which research directions may be fruitful, using relatively explicit measures of elegance and coherence. The successful use of category theory for such purposes suggests that it provides at least the beginnings of such a foundation.

References

[1] Peter Aczel. *Non-Well-Founded Sets.* Center for the Study of Language and Information, Stanford University, 1988. CSLI Lecture Notes, Volume 14.

[2] Michael Barr and Charles Wells. *Toposes, Triples and Theories.* Springer, 1985. Grundlehren der mathematischen Wissenschafter, Volume 278.

[3] Michael Barr and Charles Wells. The formal description of data types using sketches. In Michael Main, A. Melton, Michael Mislove, and D. Schmidt, editors, *Mathematical Foundations of Programming Language Semantics.* Springer, 1988. Lecture Notes in Computer Science, Volume 298.

[4] Luitzen Egbertus Jan Brouwer. Intuitionistische betrachtungen über den formalismus. *Koninklijke Akademie van wetenschappen te Amsterdam, Proceedings of the section of sciences*, 31:374–379, 1928. In *From Frege to Gödel*, Jean van Heijenoort (editor), Harvard, 1967, pages 490–492.

[5] Rod Burstall. An algebraic description of programs with assertions, verification, and simulation. In J. Mack Adams, John Johnston, and Richard Stark, editors, *Proceedings, Conference on Proving Assertions about Programs*, pages 7–14. Association for Computing Machinery, 1972.

[6] Rod Burstall and Joseph Goguen. Putting theories together to make specifications. In Raj Reddy, editor, *Proceedings, Fifth International Joint Conference on Artificial Intelligence*, pages 1045–1058. Department of Computer Science, Carnegie-Mellon University, 1977.

[7] Rod Burstall and Joseph Goguen. The semantics of Clear, a specification language. In Dines Bjorner, editor, *Proceedings, 1979 Copenhagen Winter School on Abstract Software Specification*, pages 292–332. Springer, 1980. Lecture Notes in Computer Science, Volume 86; based on unpublished notes handed out at the Symposium on Algebra and Applications, Stefan Banach Center, Warsaw, Poland, 1978.

[8] Pierre-Luc Curien. *Categorial Combinators, Sequential Algorithms, and Functional Programming.* Pitman and Wiley, 1986. Research Notes in Theoretical Computer Science.

[9] Hans-Dieter Ehrich. On the theory of specification, implementation and parameterization of abstract data types. *Journal of the Association for Computing Machinery*, 29:206–227, 1982.

[10] Hartmut Ehrig. Introduction to the algebraic theory of graph grammars. In V. Claus, Hartmut Ehrig, and Gregor Rozenberg, editors, *Graph Gramars and their Application to Computer Science and Biology*, pages 1–69. Springer, 1979. Lecture Notes in Computer Science, Volume 73.

[11] Samuel Eilenberg and Saunders Mac Lane. General theory of natural equivalences. *Transactions of the American Mathematical Society*, 58:231–294, 1945.

[12] Gian Luigi Ferrari. *Unifying Models of Concurrency.* PhD thesis, University of Pisa, 1990.

[13] Kokichi Futatsugi, Joseph Goguen, Jean-Pierre Jouannaud, and José Meseguer. Principles of OBJ2. In Brian Reid, editor, *Proceedings, Twelfth ACM Symposium on Principles of Programming Languages*, pages 52–66. Association for Computing Machinery, 1985.

[14] Kokichi Futatsugi, Joseph Goguen, José Meseguer, and Koji Okada. Parameterized programming in OBJ2. In Robert Balzer, editor, *Proceedings, Ninth International Conference on Software Engineering*, pages 51–60. IEEE Computer Society, March 1987.

[15] J.R.W. Glauert, K. Hammond, J.R. Kennaway, G.A. Papadopoulos, and M.R. Sleep. DACTL: Some introductory papers. Technical Report SYS-C88-08, School of Information Systems, University of East Anglia, 1988.

[16] Joseph Goguen. Mathematical representation of hierarchically organized systems. In E. Attinger, editor, *Global Systems Dynamics*, pages 112–128. S. Karger, 1971.

[17] Joseph Goguen. Minimal realization of machines in closed categories. *Bulletin of the American Mathematical Society*, 78(5):777–783, 1972.

[18] Joseph Goguen. Realization is universal. *Mathematical System Theory*, 6:359–374, 1973.

[19] Joseph Goguen. On homomorphisms, correctness, termination, unfoldments and equivalence of flow diagram programs. *Journal of Computer and System Sciences*, 8:333–365, 1974. Original version in *Proceedings, 1972 IEEE Symposium on Switching and Automata*, pages 52–60; contains an additional section on program schemes.

[20] Joseph Goguen. Semantics of computation. In Ernest G. Manes, editor, *Proceedings, First International Symposium on Category Theory Applied to Computation and Control*, pages 234–249. University of Massachusetts at Amherst, 1974. Also in Lecture Notes in Computer Science, Volume 25, Springer, 1975, pages 151–163.

[21] Joseph Goguen. Reusing and interconnecting software components. *Computer*, 19(2):16–28, February 1986. Reprinted in *Tutorial: Software Reusability*, Peter Freeman, editor, IEEE Computer Society, 1987, pages 251–263, and in *Domain Analysis and Software Systems Modelling*, Rubén Prieto-Díaz and Guillermo Arango, editors, IEEE Computer Society, 1991, pages 125–137.

[22] Joseph Goguen. What is unification? A categorical view of substitution, equation and solution. In Maurice Nivat and Hassan Aït-Kaci, editors, *Resolution of Equations in Algebraic Structures, Volume 1: Algebraic Techniques*, pages 217–261. Academic, 1989. Also Report SRI-CSL-88-2R2, SRI International, Computer Science Lab, August 1988.

[23] Joseph Goguen. Types as theories. In George Michael Reed, Andrew William Roscoe, and Ralph F. Wachter, editors, *Topology and Category Theory in Computer Science*, pages 357–390. Oxford, 1991. Proceedings of a Conference held at Oxford, June 1989.

[24] Joseph Goguen and Rod Burstall. CAT, a system for the structured elaboration of correct programs from structured specifications. Technical Report Report CSL-118, SRI Computer Science Lab, October 1980.

[25] Joseph Goguen and Rod Burstall. Some fundamental algebraic tools for the semantics of computation, part 1: Comma categories, colimits, signatures and theories. *Theoretical Computer Science*, 31(2):175–209, 1984.

[26] Joseph Goguen and Rod Burstall. Some fundamental algebraic tools for the semantics of computation, part 2: Signed and abstract theories. *Theoretical Computer Science*, 31(3):263–295, 1984.

[27] Joseph Goguen and Susanna Ginali. A categorical approach to general systems theory. In George Klir, editor, *Applied General Systems Research*, pages 257–270. Plenum, 1978.

[28] Joseph Goguen, Claude Kirchner, Hélène Kirchner, Aristide Mégrelis, and José Meseguer. An introduction to OBJ3. In Jean-Pierre Jouannaud and Stephane Kaplan, editors, *Proceedings, Conference on Conditional Term Rewriting*, pages 258–263. Springer, 1988. Lecture Notes in Computer Science, Volume 308.

[29] Joseph Goguen and José Meseguer. Correctness of recursive parallel nondeterministic flow programs. *Journal of Computer and System Sciences*, 27(2):268–290, October 1983. Earlier version in *Proceedings, Conference on Mathematical Foundations of Computer Science*, 1977, pages 580–595, Springer Lecture Notes in Computer Science, Volume 53.

[30] Joseph Goguen and José Meseguer. Models and equality for logical programming. In Hartmut Ehrig, Giorgio Levi, Robert Kowalski, and Ugo Montanari, editors, *Proceedings, 1987 TAPSOFT*, pages 1–22. Springer, 1987. Lecture Notes in Computer Science, Volume 250.

[31] Joseph Goguen and José Meseguer. Unifying functional, object-oriented and relational programming, with logical semantics. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477. MIT, 1987. Preliminary version in *SIGPLAN Notices*, Volume 21, Number 10, pages 153–162, October 1986.

[32] Joseph Goguen and José Meseguer. Software for the Rewrite Rule Machine. In Hideo Aiso and Kazuhiro Fuchi, editors, *Proceedings, International Conference on Fifth Generation Computer Systems 1988*, pages 628–637. Institute for New Generation Computer Technology (ICOT), 1988.

[33] Joseph Goguen, James Thatcher, and Eric Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In Raymond Yeh, editor, *Current Trends in Programming Methodology, IV*, pages 80–149. Prentice Hall, 1978.

[34] Joseph Goguen, James Thatcher, Eric Wagner, and Jesse Wright. A junction between computer science and category theory, I: Basic concepts and examples (part 1). Technical report, IBM Watson Research Center, Yorktown Heights NY, 1973. Report RC 4526.

[35] Joseph Goguen, James Thatcher, Eric Wagner, and Jesse Wright. Initial algebra semantics and continuous algebras. *Journal of the Association for Computing Machinery*, 24(1):68–95, January 1977. An early version is "Initial Algebra Semantics", with James Thatcher, IBM T.J. Watson Research Center, Report RC 4865, May 1974.

[36] Robert Goldblatt. *Topoi, the Categorial Analysis of Logic*. North-Holland, 1979.

[37] William S. Hatcher. *The Logical Foundations of Mathematics*. Permagon, 1982.

[38] Jacques Herbrand. Recherches sur la théorie de la démonstration. *Travaux de la Société des Sciences et des Lettres de Varsovie, Classe III*, 33(128), 1930.

[39] Horst Herrlich and George Strecker. *Category Theory*. Allyn and Bacon, 1973.

[40] C.A.R. Hoare and Jifeng He. Natural transformations and data refinement, 1988. Programming Research Group, Oxford University.

[41] Berthold Hoffmann and Detlef Plump. Jungle evaluation for efficient term rewriting. Technical Report 4/88, Fachbereich Mathematik und Informatik, Universität Bremen, 1988.

[42] Martin Hyland. The effective topos. In A.S. Troelstra and van Dalen, editors, *The Brouwer Symposium*. North-Holland, 1982.

[43] Robert Keller and Joseph Fasel, editors. *Proceedings, Graph Reduction Workshop*. Springer, 1987. Lecture Notes in Computer Science, Volume 279.

[44] Richard Kennaway. On 'On graph rewritings'. *Theoretical Computer Science*, 52:37–58, 1987.

[45] Joachim Lambek and Phil Scott. *Introduction to Higher Order Categorical Logic*. Cambridge, 1986. Cambridge Studies in Advanced Mathematics, Volume 7.

[46] Saunders Mac Lane. Duality for groups. *Proceedings, National Academy of Sciences, U.S.A.*, 34:263–267, 1948.

[47] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer, 1971.

[48] Saunders Mac Lane. To the greater health of mathematics. *Mathematical Intelligencer*, 10(3):17–20, 1988. See also *Mathematical Intelligencer 5*, No. 4, pp. 53–55, 1983.

[49] F. William Lawvere. Functorial semantics of algebraic theories. *Proceedings, National Academy of Sciences, U.S.A.*, 50:869–872, 1963. Summary of Ph.D. Thesis, Columbia University.

[50] Humberto Maturana and Francisco Varela. *The Tree of Knowledge*. Shambhala, New Science Library, 1987.

[51] José Meseguer and Joseph Goguen. Initiality, induction and computability. In Maurice Nivat and John Reynolds, editors, *Algebraic Methods in Semantics*, pages 459–541. Cambridge, 1985.

[52] José Meseguer and Ugo Montanari. Petri nets are monoids: A new algebraic foundation for net theory. In *Proceedings, Symposium on Logic in Computer Science*. IEEE Computer Society, 1988. Full version in Report SRI-CSL-88-3, Computer Science Laboratory, SRI International, January 1988; submitted to *Information and Computation.*

[53] Robin Milner. An algebraic definition of simulation between programs. Technical Report CS-205, Stanford University, Computer Science Department, 1971.

[54] Robin Milner. *A Calculus of Communicating Systems.* Springer, 1980. Lecture Notes in Computer Science, Volume 92.

[55] Eugenio Moggi. Computational lambda-calculus and monads. Technical Report ECS-LFCS-88-66, Laboratory for Foundations of Computer Science, University of Edinburgh, 1988.

[56] Eugenio Moggi. A category-theoretic account of program modules, 1989. Laboratory for Foundations of Computer Science, University of Edinburgh.

[57] Robert Paré and Peter Johnstone. *Indexed Categories and their Applications.* Springer, 1978. Lecture Notes in Mathematics, Volume 661.

[58] Jean Claude Raoult. On graph rewritings. *Theoretical Computer Science*, 32:1–24, 1984.

[59] Dana Scott. Continuous lattices. In *Proceedings, Dalhousie Conference*, pages 97–136. Springer, 1972. Lecture Notes in Mathematics, Volume 274.

[60] Stuart Shieber. *An Introduction to Unification-Based Approaches to Grammar.* Center for the Study of Language and Information, 1986.

[61] Michael Smyth and Gordon Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal of Computation*, 11:761–783, 1982. Also Report D.A.I. 60, University of Edinburgh, Department of Artificial Intelligence, December 1978.

[62] Andrzej Tarlecki, Rod Burstall, and Joseph Goguen. Some fundamental algebraic tools for the semantics of computation, part 3: Indexed categories. *Theoretical Computer Science*, 91:239–264, 1991. Also, Monograph PRG–77, August 1989, Programming Research Group, Oxford University.

[63] William Joseph Tracz. *Formal Specification of Parameterized Programs in* LILLEANNA. PhD thesis, Stanford University, to appear.

[64] Alfred North Whitehead. *Process and Reality.* Free, 1969.