A CATEGORICAL MANIFESTO

by

Joseph A. Goguen

Technical Monograph PRG-72
ISBN 0-902928-54-6

March 1989

Oxford University Computing Laboratory
Programming Research Group
8–11 Keble Road
Oxford OX1 3QD
England

Oxford University Computing Laboratory
Programming Research Group
8–11 Keble Road
Oxford OX1 3QD
England

Electronic mail: goguen@uk.ac.oxford.prg (JANET)

A Categorical Manifesto*

Joseph A. Goguen

Summary

This informal paper tries to motivate the use of category theory in computing science by giving heuristic guidelines for applying five basic categorical concepts: category, functor, natural transformation, adjoint, and colimit. Several examples and some general discussion are given for each concept, and a number of references are cited, although no attempt has been made for completeness. Some additional categorical concepts and suggestions for further research are also mentioned. The paper concludes with a brief discussion of some implications for foundations.

*Supported by Office of Naval Research Contracts N00014-85-C-0417 and N00014-86-C-0450, NSF Grant CCR-8707155, and a gift from the System Development Foundation.

Contents

0 Introduction

Among the reasons why a computing scientist might be interested in category theory are that it can provide help with the following:

- *Formulating definitions and theories.* In fields that are not yet very well developed, like computing science, it often seems that formulating basic concepts is the most difficult part of research. The five guidelines given below provide relatively explicit measures of elegance and coherence that can be helpful in this regard.

- *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 nice 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.* Finding similar structures in different areas suggests trying to find further similarities. For example, an analogy between Petri nets and the λ-calculus might suggest looking for a closed category structure on nets (as in [46], which seems to open an entirely new approach to concurrency).

- *Formulating conjectures and research directions.* For example, if you have found an interesting functor, you might be well advised to investigate its adjoints.

- *Dealing with abstraction and representation independence.* In computing science, abstract viewpoints are often better, 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 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.

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 generalized 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 any such 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 foes. Perhaps what this phrase suggests to both camps 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 theoretical foundations, and the relatively poor guidance for discovering elegant and coherent theories that they provide. Section 6 discusses this further.

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

This note presents five guidelines for using category theory, each with some general discussion and some specific examples. There is no claim to originality, since I believe the underlying intuitions are shared by essentially all workers in category theory, although they have been understandably reluctant to place such dogmatic assertions in textbooks or other written documents[1]. The five guidelines are necessarily imprecise, and will seem exaggerated if taken too literally, since 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, since the purpose of this note is motivational, and the definitions can be found in any textbook. Thus, it is necessary to use some text in connection with this note. Unfortunately, no existing text is ideal for computing scientists, but perhaps that by Goldblatt [31] comes closest. The classic text by Mac Lane [41] is warmly recommended for those with sufficient mathematics background, and Herrlich and Strecker's book [34] is admirably thorough; see also [2] and [39].

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 dogma that in order to understand a structure, it is necessary to understand the morphisms that preserve it. Indeed, many category theorists feel that the morphisms are more important than the objects, since they reveal what the structure really is. Moreover, the category concept can be defined using only morphisms. It is the bias of modern Western language and culture towards objects, rather than towards relationships, that assigns precedence to objects over morphisms (see [44] for some related discussion). Now some examples to illustrate this dogma:

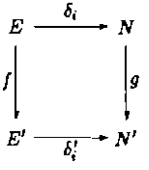
1.1 **Sets.** If we take sets to be objects, then their morphisms are clearly going to be functions. A set morphism, however, is not just a set of ordered pairs, but must also specify particular source and target sets. This is consistent with practice in computation theory which assigns types to functions. 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

[1] As far as I know, the first such attempt is my own, given in [29], which contains the first four guidelines given here; the present note can be seen as an expansion of that early attempt. The only other attempt that I know is due to Lambek and Scott [39], who give a number of "slogans" in a similar style.

is that any two sets of ordered pairs can be composed to yield a third. The category *Set* of sets embodies a contrary point of view, that each function has a domain in which its arguments are meaningful, and a codomain in which its results are meaningful. (See [31] for further discussion of these points.)

1.2 **Relations.** Just as with functions, it seems desirable to take the view that the composition of relations is only meaningful within certain given domains. 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 *Rel*, of which *Set* 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 $\partial_0, \partial_1 : E \to N$ which give the **source** and **target** of each edge, respectively. Since 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 *Set* for $i = 0, 1$:

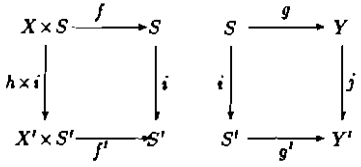


To show that we have a category *Graph* of graphs, we must show that a composition of two such morphisms is another, and that a pair of identity functions satisfies the diagrams and also serves as an identity for composition.

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 $Pa(G)$. Details may be found in [41], [29], [16], and many other places.

1.5 **Substitutions.** Two key attributes of a substitution are the set of variables for which it substitutes, and the set of variables that occur in what it substitutes. Thus, substitutions naturally have source and target objects, each a set of variables. Clearly there are identity substitutions for each set of variables (substituting each variable for itself), and the composition of substitutions is associative when defined. See [19] for much more on this example; in fact, [19] can be used as a primer on category theory, motivated by just this example and its many applications.

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



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 **Aut** of automata, and increase our confidence in the correctness of the definitions. See [15].

1.7 **Theories.** In his 1963 thesis [43], F.W. Lawvere developed a very elegant approach to universal algebra, in which an *algebraic theory* is a category **T** whose morphisms correspond to equivalence classes of terms, and whose objects indicate the variables involved in these terms, much as in Example 1.5 above. In this approach, the objects of a theory are closed under products (products are defined in Example 4.1 below). Although Lawvere's original development was unsorted, it easily extends to the many-sorted case; [19] gives a relatively concrete and hopefully readable account of these ideas for computing scientists, with many applications, following the approach indicated in Example 1.5 above. Lawvere theories have been extended in many other ways, including the so-called "sketches" by Ehresmann, Gray, Barr, Wells, and others; for example, see [3].

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 $f: A \to B$ is an **isomorphism** in a category **C** iff there is another morphism $g: B \to A$ in **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 formalized:

unless it is satisfied, the objects, or the morphisms, or both, are wrong. This principle is so pervasive that isomorphic objects are very 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" [28].

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 diagramatic style of reasoning about equality of compositions. Since it is valid for diagrams in any category whatever, this proof style is very widely applicable; for example, it applies to substitutions (see 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 example of this occurs in Example 1.3, to prove that a composition of two graph morphisms is another graph morphism.

2 Functors

The second dogma says:

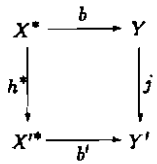
> *To any construction on structures of one species, say widgets, yielding structures of another species, say whatsits, there corresponds a functor from the category of widgets to the category of whatsits.*

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 these 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 formalized. Of course, functorality 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 coustruction gives a functor from the category of sets to the category of monoids, with a function $f: X \to Y$ inducing $f^*: X^* \to Y^*$ by sending Λ to Λ, and sending $x_1 \ldots x_n$ to $f(x_1) \ldots f(x_n)$.

2.2 **Behaviors.** Given an automaton $A = (X, S, Y, f, g)$, its *behavior* 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 behaviors. 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



commutes in **Set**. Denote this category **Beh** and define B : **Aut** $\to$ **Beh** 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 our previous definitions. See [15].

2.3 **Algebras.** In the Lawvere approach to universal algebra [43], an algebra is a functor from a theory T to **Set**. Here, "construction" takes the meaning of "interpretation": the abstract structure in T is interperted (i.e., constructed) concretely in **Set**; in particular, these functors must preserve products.

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. In model theory, the whatsit underlying a widget is called a "retract." Notice that a ring (with ideutity) is a monoid in *two different* ways, one for its additive structure and one for its multiplicitive structure.

2.5 **Categories.** Of course, the (small) categories also form a category, with functors as morphisms. It is denoted **Cat**.

2.6 **Diagrams and the Path Category Construction.** The construction in Example 1.4 of the category $\mathbf{Pa}(G)$ of all paths in a graph G gives rise to a functor **Pa**: **Graph** $\to$ **Cat** from graphs to categories. Then a diagram in a category C, with shape a graph G, is a functor $D : \mathbf{Pa}(G) \to \mathrm{C}$. It is conventional to write just $D\colon G \to \mathrm{C}$, and even to call D a "functor," because $D : \mathbf{Pa}(G) \to \mathrm{C}$ is in fact fully determined by its restriction to G, which is a graph morphism; see Example 4.2 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 **Rel** 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 $P(e)\colon P(n) \to P(n')$ 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

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

This approach originated in Burstall [5]. Techniques that allow programs to have *syntax* as well as semantics are described in [16][2]: A **program scheme** is a functor $P : G \to \mathsf{T}$ into a Lawvere theory T "enriched" with a partial order structure on its morphism sets $\mathsf{T}(A, B)$ (the reader familiar with 2-categories should notice that this makes T a 2-category). Semantics for statements then arises by giving a functor $A : \mathsf{T} \to \mathbf{Rel}$, that is, an interpretation for T, also called a T-algebra. The semantics of a program is then computed by the above formula for the composition $P.A : G \to \mathbf{Rel}$. There seems to be much more research that could be done in this area. For example, [24] 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.

3 Naturality

The third dogma says:

> *To each natural relationship between two functors $F, G : \mathsf{A} \to \mathsf{B}$ corresponds a natural transformation $F \Rightarrow G$ (or perhaps $G \Rightarrow F$).*

Although this looks like a mere definition of the phrase "natural relationship," 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 formalize the notion of an equivalence of homology theories (whatever they are), 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 [43], algebras appear as functors, and so we should expect homomorphisms to appear as natural transformations; and indeed, they do.

3.2 **Natural Equivalence.** A natural transformation $\eta : F \Rightarrow G$ is a natural equivalence iff each $\eta_A : FA \to GA$ is an isomorphism. This is the natural notion of isomorphism for functors, and is equivalent to the existence of $\nu : G \Rightarrow F$ such that $\nu.\eta = 1_F$ and $\eta.\nu = 1_G$. This concept specializes to isomorphism of algebras, and is also exactly the concept that motivated Eilenberg and Mac Lane.

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

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 that take more than one argument. However, such theories still admit algebras, which are functors into $\mathbf{Set}$, 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 leads to some general techniques for developing correct programs [35]. It would be interesting to extend this to more expressive forms of Lawvere theory (such as many-sorted theories or sketches), and to the more general data representations studied in the abstract data type literature (e.g., [28, 9]).

3.4 **Program Homomorphisms.** Since Example 2.7 suggests that programs are functors, we should expect to get a kind of program homomorphism from a natural transformation between programs. Indeed, Burstall [5] shows that a weak form of Milner's program simulations [47] arises in just this way. [16] generalizes this to programs with different shapes, and maps from edges to paths, by defining a **homomorphism** from $P_0 : G_0 \to \mathbf{C}$ to $P_1 : G_1 \to \mathbf{C}$ to consist of a functor $F : G_0 \to \mathbf{P}a(G_1)$ and a natural transformation $\eta : P_0 \to F.P_1$; some theory and applications are also given, including techniques for proving correctness, termination, and non-trivial equivalences by unfolding programs into equivalent infinite trees.

4 Adjoints

The fourth dogma says:

> *Any canonical construction from widgets to whatsits is an adjoint of another functor, from whatsits to widgets.*

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 a certain condition. 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

4.1 **Products and Sums.** One nice achievement of category theory is to give general definitions for previously vague terms like "product" and "sum" (although sums are usually called "coproducts"). For example, the Cartesian product of sets is a functor $\mathbf{Set} \times \mathbf{Set} \to \mathbf{Set}$. The general definitions make sense in any category $\mathbf{C}$, and characterize the construction uniquely (up to isomorphism) if it exists. Let $\Delta : \mathbf{C} \to \mathbf{C} \times \mathbf{C}$ be the "diagonal" functor, sending an object C in $\mathbf{C}$ to the pair (C, C), and sending a morphism $c : C \to C'$ in $\mathbf{C}$ to $(c, c) : (C, C) \to (C', C')$ in $\mathbf{C} \times \mathbf{C}$. Then $\mathbf{C}$ has products iff Δ has a right

adjoint, and has sums iff Δ has a left adjoint. This is a beautifully simple way to formalize mathematical concepts of basic importance that were previously only understood informally (due to Mac Lane [40]).

4.2 **Freebies.** Another beautifully simple formalization gives a general definition of "free" constructions: they are the left adjoints of forgetful functors. For example, the path category functor $Pa: Graph \to Cat$ of Example 2.6 is left adjoint to the forgetful functor $Cat \to Graph$, and thus may be said to give the free category over a graph.

4.3 **Minimal Realization.** An automaton (X, S, Y, f, g) is *reachable* iff its function $\overline{f}: X^* \to S$ is surjective. Let $\mathbf{A}$ denote the subcategory of $\mathbf{Aut}$ whose objects are reachable and whose morphisms (i, j, k) have i surjective. Then the restriction $B: \mathbf{A} \to \mathbf{Beh}$ of $B: \mathbf{Aut} \to \mathbf{Beh}$ to $\mathbf{A}$ has a right adjoint which gives the minimal realization of a behavior [15]. Since right adjoints are uniquely determined, this provides a convenient abstract characterization of minimal realization. Moreover, this characterization extends to, and even suggests, more general minimal realization situations.

4.4 **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 [43].

4.5 **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 [39]. This connection has been used, for example, as a basis for the efficient compilation of higher order functional languages [8]. An advantage is that optimization techniques can be proved correct by using purely equational reasoning.

5 Colimits

The fifth dogma says:

> *Given a category of widgets, the operation of putting a system of widgets together to form some super-widget corresponds to taking the colimit of the diagram of widgets that shows how to interconnect them.*

At least for me, this dogma first appeared in the context of General Systems Theory [23]. It is worth remarking that, generalizing Example 4.1, colimits over the diagrams of a fixed shape G (a graph) give a functor that is right adjoint to a (suitably generalized) diagonal functor. Now some examples:

5.1 **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 logaritbm" 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 [12], Eqlog [25], and FOOPS [26] extend this notion of generic module to functional, logic (i.e., relational), and object oriented programming, respectively. It has even been applied to Ada [18].

5.2 **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 [37, 27, 13], and colimits seem to be quite useful here [10, 52, 38]. Graph rewriting is also important for the *unification grammars* that are now popular in linguistics [53, 19] There seem to be many opportunities for further research in these areas.

5.3 **Initiality.** The simplest possible diagram is the empty diagram. Its colimit is an *initial object*, which 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. It is also worth mentioning that universality can be reduced to initiality (in a comma category), and hence so can colimits.

5.4 **Initial Model Semantics.** It seems remarkable that initiality is so very useful in computing science. Beginning with the formalization of abstract syntax as an initial algebra [17], initiality has been applied to an increasing range of fundamental concepts, including induction and recursion [30, 45], abstract data types [28], computability [45], and model theoretic semantics for functional [12], logic (i.e., relational), combined functional and relational, and constraint logic [25] 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 formalizing, generalizing, and smoothing out, the classical Herbrand Universe construction [33], and it seems likely that much interesting work remains to be done along these lines.

Research in General Systems Theory also suggests a dual dogma, that the *behavior* of a system is given by a limit construction [23]. For example, this can be used to justify the formula in Example 2.7, and to explain the sense of "unification" in so-called unification grammars [19].

6 Further Topics

Although they are particularly fundamental, the five 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.

6.1 **Comma Categories.** Comma categories are another basic construction that first appeared in Lawvere's thesis [43]. They tend to arise when morphisms are used as objects. Examples 1.3, 2.2, 5.3, and 5.4 in this paper can all be seen as comma categories. Viewing a category as a comma category makes available (for example) some very general results to prove the existence of limits and colimits [21].

6.2 **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 [20], [22], [35], [50] and other places.

6.3 **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 [46] has already been mentioned, and [14] suggests that monoidal categories may provide a general approach to understanding the relationships among the many different theories of concurrency.

6.4 **Indexed Categories.** A strict indexed category is just a functor $\mathbb{B}^{op} \to \mathcal{C}at$. [54] shows that there are many such categories in computing science, and gives some general theorems about them, including simple sufficient conditions for completeness of the associated "Grothendieck" category. Moggi [50] 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 applied in foundational studies [51].)

6.5 **Kleisli Categories.** Another way to generalize Lawvere theories is to view an arbitrary adjunction as a kind of theory. So-called monads (or triples) are an abstraction of the necessary structure, and the Kleisli category over a monad gives the category of free algebras [41]. Again, there are surprisingly many examples. [19] (in effect) takes the Kleisli category itself as a theory, and then shows that many different problems of unification (that is, of solving systems of equations) can be naturally formulated as finding equalizers in Kleisli categories. Moggi [49] uses Kleisli categories to get an abstract notion of "computation" which gives rise to many interesting generalizations of the λ-calculus.

6.6 Topoi. A profound generalization 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 a relationships to algebraic geometry, computing science, and intuitionistic logic [31, 2, 36].

7 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 successfully constructed all the objects of greatest interest in contemporary 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 Generalized 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 $S_1, S_2, S_3, \ldots$ 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 sets, to model various phenomena in computation, including communicating processes in the sense of Milner [48]. 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 fonnd all of mathematics.

Set theoretic foundations have also failed to account for mathematical practice in certain areas, such as category theory itself, and moreover have enconraged research into areas that have little or nothing to do with mathematical practice, such as large cardinals. (Mac Lane [42] gives some lively discussion of these issues, and [32] gives 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 issne no longer seems as urgent as it did, because no new paradoxes have been discovered for a long time.

This note 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 varions branches of mathematics and its applications. I now 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 of the various areas of mathematics and its applications, as well as relationships among them.*

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 new sense are much more useful, since they can suggest which research directions are most fruitful, and can test the results of research using relatively explicit measures of elegance and coherence. The successful use of category theory for such purposes suggests that it provides at least the beginning 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-Verlag, 1984. 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-Verlag, 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 University Press, 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-Verlag, 1980. Lecture Notes in Computer Science, Volume 86.

[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-Verlag, 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] Kokichi Futatsugi, Joseph Goguen, Jean-Pierre Jouannaud, and José Meseguer. Principles of OBJ2. In Brian Reid, editor, *Proceedings, 12th ACM Symposium on Principles of Programming Languages*, pages 52–66. Association for Computing Machinery, 1985.

[13] 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.

[14] Joseph Goguen. Towards a general understanding of the semantics of concurrent computation. Given as a lecture at ICOT Workshop on Concurrent Programming, 5 December 1988, Tokyo, Japan; paper in preparation.

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

[16] 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.

[17] 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, Lecture Notes in Computer Science, Volume 25, Springer-Verlag, 1975, pages 151-163.

[18] Joseph Goguen. Reusing and interconnecting software components. *Computer*, 19(2):16–28, February 1986. Also reprinted in *Tutorial: Software Reusability*, Peter Freeman, editor, IEEE Computer Society Press, 1987, pages 251-263.

[19] 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*. Academic Press, to appear, 1989. Also, Technical Report SRI-CSL-88-2R2, SRI International, Computer Science Lab, August 1988.

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

[21] 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.

[22] 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.

[23] 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.

[24] 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-Verlag Lecture Notes in Computer Science, Volume 53.

[25] 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-Verlag, 1987. Lecture Notes in Computer Science, Volume 250.

[26] Joseph Goguen and José Meseguer. Unifying 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 Press, 1987. Preliminary version in *SIGPLAN Notices*, Volume 21, Number 10, pages 153-162, October 1986.

[27] Joseph Goguen and José Meseguer. Software for the rewrite rule machine. In *Proceedings, International Conference on Fifth Generation Computer Systems 1988*, pages 628–637. Institute for New Generation Computer Technology (ICOT), 1988.

[28] Joseph Goguen, James Thatcher, and Eric Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. Technical Report RC 6487, IBM T.J. Watson Research Center, October 1976. Appears in *Current Trends in Programming Methodology, IV*, Raymond Yeh, editor, Prentice-Hall, 1978, pages 80-149.

[29] 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. Research Report RC 4526.

[30] 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.

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

[32] William S. Hatcher. *Foundations of Mathematics.* W.B. Saunders, 1968.

[33] 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.

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

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

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

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

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

[39] Joachim Lambek and Peter Scott. *Introduction to Higher Order Categorical Logic.* Cambridge University Press, 1986. Cambridge Studies in Advanced Mathematics, Volume 7.

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

[41] Saunders Mac Lane. *Categories for the Working Mathematician.* Springer-Verlag, 1971.

[42] 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.

[43] 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.

[44] Humberto Maturana and Francisco Varela. *The Tree of Knowledge.* Shambhala, 1987.

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

[46] 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, 1988. Full version in Technical Report SRI-CSL-88-3, Computer Science Laboratory, SRI International, January 1988; submitted to *Information and Computation.*

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

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

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

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

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

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

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

[54] Andrzej Tarlecki, Rod Burstall, and Joseph Goguen. Some fundamental algebraic tools for the semantics of computation, part 3: Indexed categories. *Theoretical Computer Science,* 1989. To appear.