ANALOGY CATEGORIES, VIRTUAL MACHINES, AND STRUCTURED PROGRAMMING

B.R. Gaines
Man-Machine Systems Laboratory,
Dept. of Electrical Engineering Science,
University of Essex, Colchester, U.K.

Abstract This paper arises from a number of studies of machine/problem relationships, software development techniques, language and machine design. It develops a category-theoretic framework for the analysis of the relationships between programmer, virtual machine, and problem that are inherent in discussions of "ease of programming", "good programming techniques", "structured programming", and so on. The concept of "analogy" is introduced as an explicatum of the comprehensibility of the relationship between two systems. Analogy is given a formal definition in terms of a partially ordered structure of analogy categories whose minimal element is a "truth" or "proof" category. The theory is constructive and analogy relationships are computable between defined systems, or classes of system. Thus the structures developed may be used to study the relationships between programmer, problem, and virtual machine in practical situations.

1. Introduction

There has long been a folk-lore of computing comprising moralistic fables (ESPOL and the Cactus Stack), mysterious creatures (the "good" programmer) and dark rites ("structured programming"), all concerned with value judgements about machines, problems and programmers, and their interrelationships. Like all real folk-lore this wealth of material cannot be dismissed - it provides the only constructive approaches to many problems central to computer systems engineering. And yet is is difficult to incorporate it in computer science because:

(a) it is evaluative rather than descriptive - not, "technique A exists", but, "technique A is better than technique B";

(b) as essential human element is often involved - not, "modular programs run better", but "modular programming techniques encourage programmers to produce better results".

These sources of difficulty, both involving subjective elements, have tended to undermine attempts to take a scientific approach to software development, or virtual machine design, and to make the results of studies in these areas to consist of isolated techniques or authoritarian dogma.

One effect of these problems has been to emphasize research on software production techniques that minimize human involvement, such as automatic program verification [1] which evaluates only in terms of 'correctness', or non-imperative, assertional languages [2] and theorem-proving [3] where programming is reduced to problem-description. However, the rigour of approach possible in these areas comes only because they avoid, rather than resolve, the problems stated above. The concept of program-proving is one component of "structured programming" but it does not contribute in itself to the actual process of structuring the problem to be suitable for algorithmic solution on a particular virtual machine. We cannot avoid the human component in terms such as "good programming techniques", "good machine design", and so on - terms which we all understand as going way beyond the sheer physical evaluation of correctness, speed, cost, etc.

It is the contention of this paper that both the problems stated above can be overcome and that a rigorous mathematical foundation can be established for the analysis and development of program development techniques, virtual machine design, and so on. The formulation proposed in this paper has the advantage of being constructive and leading to evaluations that can be computed in practical situations. The previously formalized concept of program verification plays a key role as a pivot for a far wider formalization of problem/programmer/machine relationships, in which both imperative and assertional languages appear as natural elements.

The basis for the formulation is the concept of an analogy relation as an explicatum of the comprehensibility of the relationship between two systems. The use of category theory enables the analogy relation to be formally defined independently of any particular structures for the two systems, and hence avoids the pre-supposition of theories of human cognitive skills, program structures, or the representation of problems. The application of the theory requires the relevant categories to be defined (in terms of automata [4,5], Petri nets [6], or lattices of flow diagrams [7,8], etc.), but the basic theory itself is independent of changes in our techniques for system representation. It turns out that the possible analogy relations between two systems form a natural and significant partial order (in fact a semi-lattice) and are finite in number when the two systems are themselves finite. It is these two properties, coupled with their psychological significance, which make analogy relations a practical explicatum of many of the concepts of structured programming.

The next section of this abstract is concerned with presenting the problems discussed in terms of a three-part relationship between programmer, problem and machine. Section 3 is a formal presentation and discussion of a category-theoretic formulation of analogy relations. The final section is concerned with how the results obtained may be applied (this paper presents work in progress and it is expected that the actual paper and presentation will contain more exemplars than can be given at present).

2. Machines, Problems and Programmers

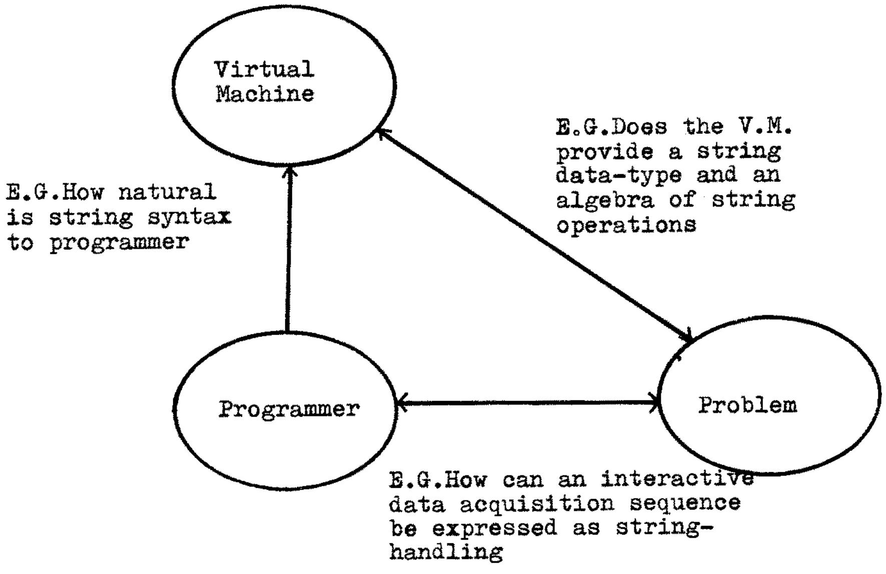
Virtual Machine
E.G.Does the V.M. provide a string data-type and an algebra of string operations
E.G.How natural is string syntax to programmer
Programmer
Problem
E.G.How can an interactive data acquisition sequence be expressed as string-handling

Figure 1 The Three-Part Relationship Between Virtual Machine, Programmer and Problem (with examples)

The obvious relationship to analyse in studying ease of programming is that between virtual machine and problem. However, this leaves the human component implicit in the evaluation, and a better basis for analysis is that of Fig. 1 which shows the full three-part relationship between virtual machine, problem, and programmer. Introducing the programmer explicitly and emphasizing the symmetry of the three separate relationships is important in enabling us to distinguish, for example, between something being "easier for the programmer" because: (a) it contributes to making the virtual machine intrinsically easier to use and understand; (b) it contributes to structuring the problem in a more comprehensible form; (c) it makes for a simple relation between problem and virtual machine which it is easy to express as a program. These possibilities are readily confounded - languages are both problem-orientated and programmer-orientated in their facilities and either aspect may make a contribution to ease of programming. Published discussions of structured programming [9] move freely between these three possibilities, commenting on language facilities which make for readable programs (the machine/programmer relationship), the structured fragmentation of problems for ease of understanding (the problem/programmer relationship), program verification (the problem/machine relationship), and so on.

Fig. 2 shows how the basic triangle of Fig. 1 iterates naturally to portray the tree of virtual machines [10,11] found on most systems. The usual hierarchy of the machines themselves is apparent, but its supplementation by the explicit incorporation of the programmer/machine relationships places new emphasis on the decoupling action of a virtual machine structure - the problem of the programmer at one level is the virtual machine of the next lower level, and there are no direct linkages between levels. One obvious question to ask in terms of Fig. 2 is whether a programmer/problem pair is being linked in at the appropriate node in the hierarchy, e.g. if, for some reason, the fluid dynamicist shown in Fig. 2 was tackling problems requiring high-speed bit manipulation, or list-processing, he might be better off linked to VM_{n+2} (intermediate language) or VM_{n+3} (LISP), respectively. That is, it raises the question of the VM node that has <u>greatest analogy to the problem structure</u>. However, on informing our errant programmer of these preferred alternatives, we are roundly informed that he finds the intermediate language too vast to remember and the LISP syntax too weird for words - FORTRAN is to him a natural language and he is sticking to it. That is, there is another question as to the VM node that has <u>greatest analogy to the programmer's (current) cognitive structure</u>!

The term "analogy" used in the preceding discussion has obvious colloquial connotations, but unless the meaning of the term can be defined more precisely, preferably operationally and quantitatively, the arguments must remain at their usual informal level. I first attempted to develop a rigorous explicatum for the concept when working on programmable digital differential analysers (DDA's) and attempting to classify problems in terms of the appropriate computing techniques [12]. In solving differential equations it is clear that the DDA has not only advantages in speed but also in ease of use. The psychological advantage arises because it is an <u>analog</u> computer whose structure closely resembles that of the differential equations it solves. The important psychological relationship between analogy and ease of use is explored in Ref. 1 where a tentative formulation in terms of category theory is proposed.

Although developed in a framework where it is fairly obviously appropriate, once abstracted this concept of analogy proved capable of wider extension to language and machine analysis and design. For example, two minicomputer designs provided a contrast between the earlier machine aimed at high packing density of programs (a major technical objective in microcomputers where store costs dominate) and the later machine aimed at ease of program development. The stark contrast between the requirement for detailed hand-coding and impossibility of compilation of the former, and the natural relationship to algebraic language of the latter, placed analogy in the role of another technical factor that could be traded against, for example, program packing density. There was a strong incentive to quantify "analogy" in such a way that these trade-offs could be clearly expressed. A possible quantification, based on the tentative category-theoretic concepts of Ref. 1 but now worked out in detail, is given in the following section. It turns out to be surprisingly straightforward and capable of direct application.

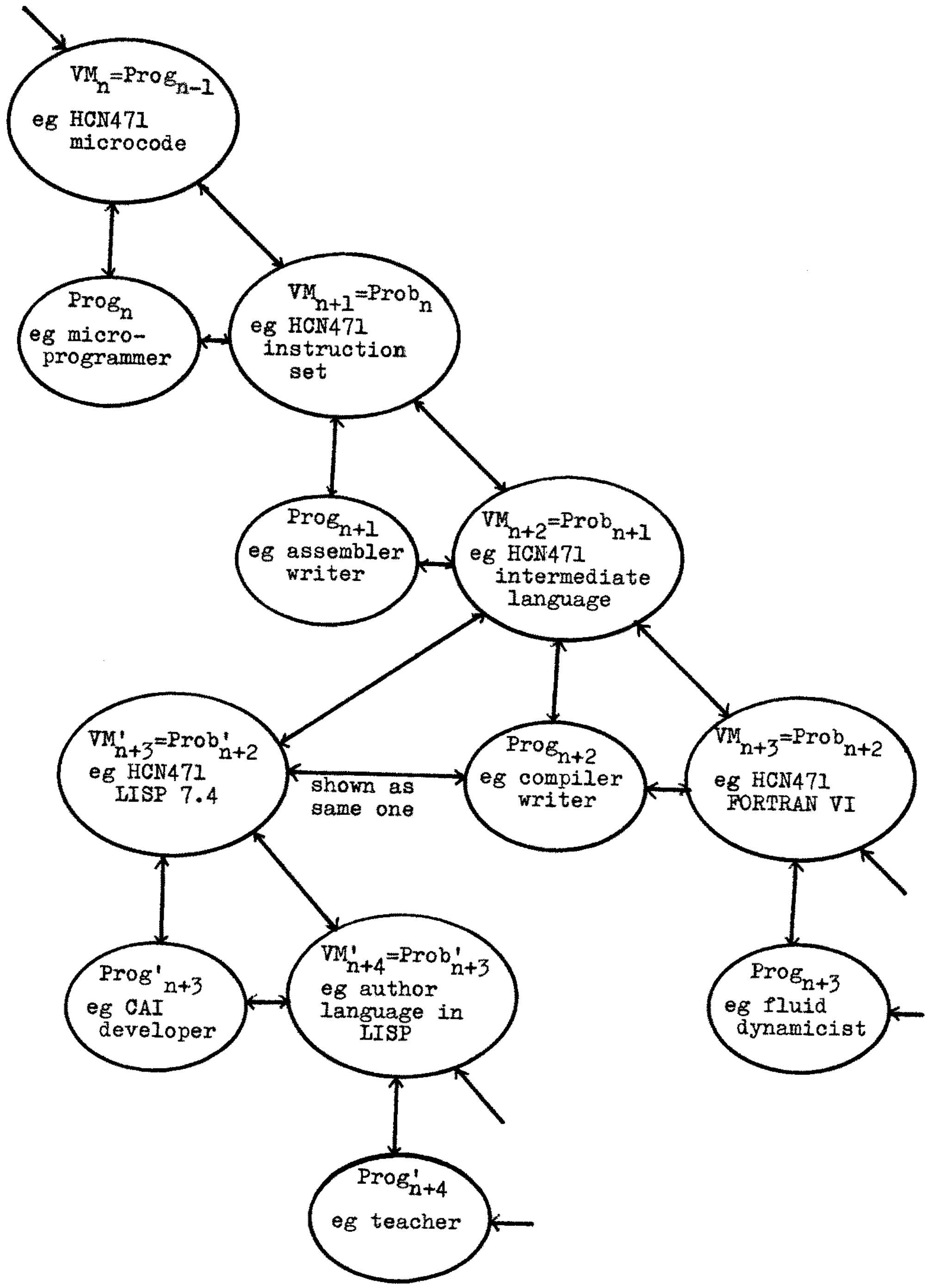

Figure 2 The Hierarchy of Virtual Machines and Programmers

3. A Category-Theoretic Formulation of Analogy

If we had tried to formulate the concept of an analogy relation a decade ago we would have been forced to frame it in terms of particular algebraic or topological structures. For the machine, a finite automaton structure would have been obvious. For the programmer or problem, however, any single structure would have imposed severe restrictions on the generality of the results and left them open to criticisms which applied only to the specific structures chosen to model human cognitive processes, or problem specifications, and not to the notion of analogy itself.

A category-theoretic framework for a theory of analogy avoids these problems. By representing the machine, programmer, and problem as arbitrary categories, the way is left open for any particular structure to be postulated for any one of them, and for the accepted structures to change with our states of knowledge and technology without affecting the fundamental concept of analogy. In addition, even if the basic structures we use remain unaltered, the use of category theory enables us to cope with changes in emphasis and significance - we may wish to examine the analogy between a particular problem and a particular program, or between a class of problems and a class of programs - we may wish to specify either a particular value or a particular function as a result to be verified. A category can be highly specific, e.g. a single discrete set, or highly general, e.g. a class of algebras, and it can express constraints upon both objects and functions.

This leads naturally into our first postulate:

Postulate I A system can be represented by a category.

This is, perhaps, immediately acceptable for virtual machines, acceptable on trust for problems, but dubious for programmers! The first two cases are adequate for many important results, and if programmer is replaced by, 'cognitive model of programmer', then the third case becomes more reasonable. Goguen's papers on category theory applied to the semantics of computation [8,13], system structure and behaviour [14,15], and human and artificial cognitive processes [16], present the case for this postulate far better than any arguments here.

The next question is how may we compare two systems (categories) for an analogy between them ? To get some idea of what is involved it is useful to have some informal specific category in mind, say that of automata [14,15]. The notion of isomorphism, or any kind of morphism, between the categories is not useful because in general we expect each to have structure not reflected in the other - an analogy is a partial correspondence - one automaton may transit many states during one transition in the other, and vice versa, but some states of each can be put into mutual correspondence. Since we cannot map directly from one system to the other we introduce a "correspondence" category that maps onto each, and ensure that these mappings are non-trivial by requiring them to be faithful functors. A faithful functor has important structure correspondence properties in that it carries commutative diagrams in one category into commutative diagrams in the other, in both directions.

Despite this restriction however our structure, like all partial correspondence concepts, is as yet very weak and allows for many trivial "correspondence categories". We strengthen it by introducing a key concept, that of a "truth" category, which is a correspondence category with the minimal structure sufficient to express the essence of one of the other two categories. For example, suppose one of our categories is essentially a description of a process for calculating tax due, and our other category (which we shall call the "model") is essentailly a computer program to perform this calculation. Then the truth category might represent a simple input/output map of data in and results out, i.e. we are not interested in how the original calculation was done and do not want this to be reflected in the program - all we want are correct final results for given data.

Note that the redundancy in the problem specification will probably be not only in structure but also in the domains of data - the domains in the truth category will tend to be smaller than the implied domains in the problem specific-ation (and the actual domains in the computer program). The truth category is the minimal structure that we wish to reflect from the problem category through the truth category into the modal category, and vice versa. It clearly forms the basis for program verification and may be termed a "proof" category when the main categories are a problem and a program.

Postulate II A truth category having a faithful functor to each of a category and its model can adequately represent all that we mean by a "correct", or "significant", or "adequate", or "true", analogy.

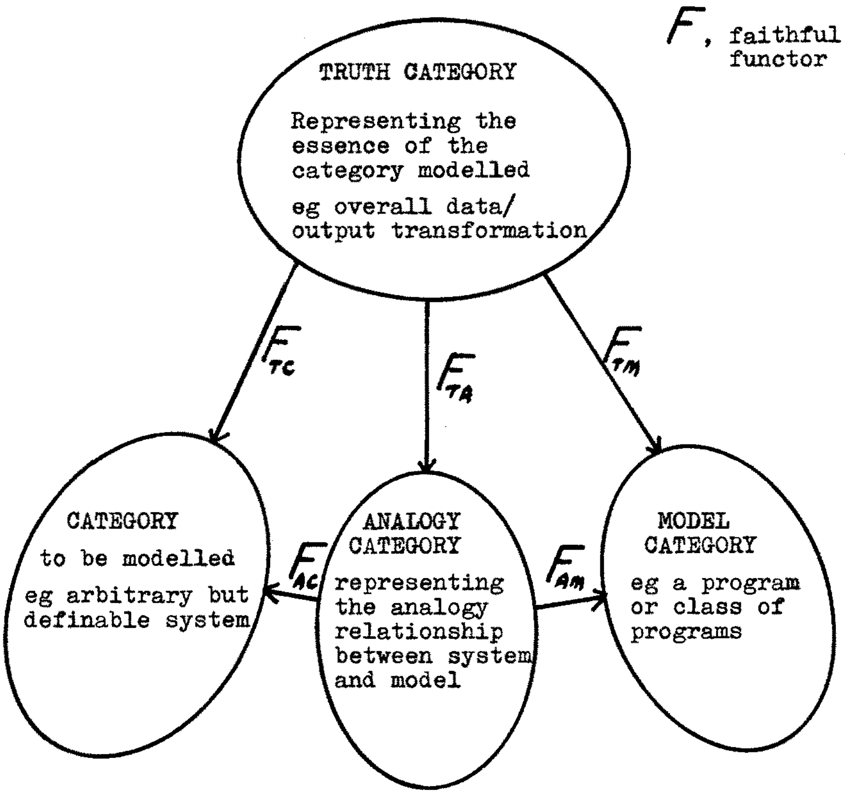
F, faithful functor
TRUTH CATEGORY
Representing the essence of the category modelled
eg overall data/ output transformation
F_{TC}
F_{TA}
F_{TM}
CATEGORY
to be modelled
eg arbitrary but definable system
ANALOGY CATEGORY
representing the analogy relationship between system and model
MODEL CATEGORY
eg a program or class of programs
F_{AC}
F_{AM}

Figure 3 Diagram Defining the Analogy Category Between a System and its Model

We now have sufficient structure to formulate the concept of an "analogy category", or just "analogy". It is a correspondence category that makes the diagram of Fig. 3 commute, i.e. the faithful functors from the truth category factor through the analogy category. Hence the functors from the analogy category reflect all properties reflected by the truth category, together with certain others that the category and its model have in common but which go beyond those strictly required by the truth category. It is of course just these other properties which make the difference between the analogy for addition, say, offered by a universal Turing machine and that offered by a digital computer - at truth level the Turing machine is everything that the computer can be.

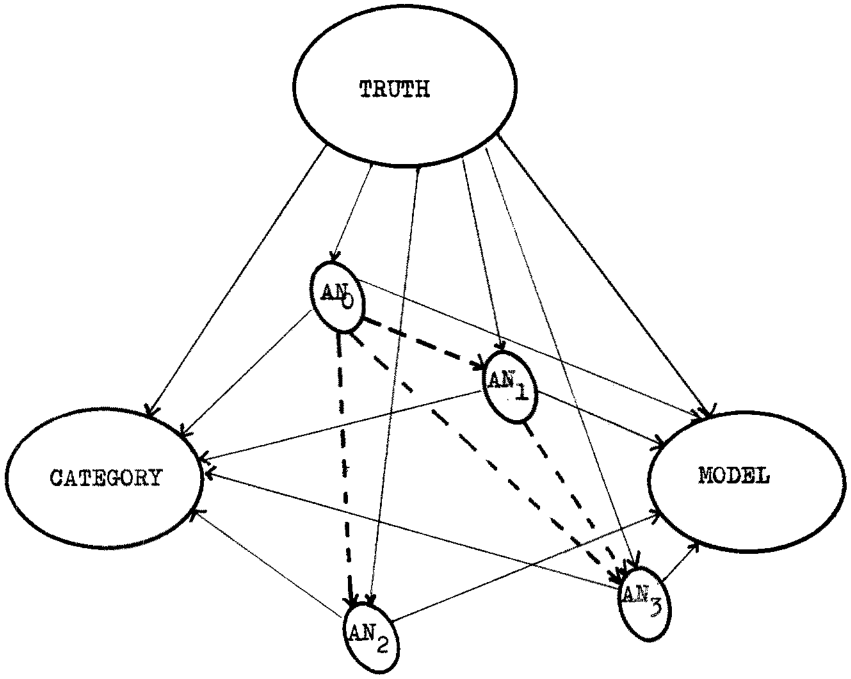

Figure 4 A Semi-Lattice of Analogies

The arrows are faithful functors: $\longrightarrow$ necessary

$- - \rightarrow$ possible

There can clearly be many analogy categories for a given category/truth/model (CTM) triple, but the direction and faithfulness of the functors guarantee that the analogy categories are "smaller" than either the category or its model. Fig. 4 shows a set of four analogies, AN_0, AN_1, AN_2 and AN_3. Each necessarily has the prescribed triple of arrows connecting it to the CTM triple. However, there may also be faithful functors between the analogies themselves, and these define an important relation between analogies. Because the existence of faithful functors is reflexive, asymmetric and transitive, the relation induced is a partial order, and we shall write:

$$AN_n \geq AN_m \iff F: AN_m \xrightarrow{\text{faithful}} AN_n$$

where AN_n and AN_m are analogy categories. The relation is in fact somewhat stronger since we can show that least upper bounds, if they exist, are unique, and greatest lower bounds always exist and are unique (truth is a universal lower bound), and hence analogies form a lower semi-lattice.

It is this semi-lattice structure that forms the richest component of our formulation of analogy - it gives a rigorous explicatum to the concept of one structure being more analogous than another and it ensures that if two analogies cannot be compared directly there is a unique common analogy (their glb) which expresses their maximum mutual relationship.

Postulate III The semi-lattice ordering of analogy categories adequately represents what we mean by one analogy being "more comprehensive", "closer", or "more detailed", than another.

The role of the truth category may now be seen as a constraint ensuring the relevance of an analogy (our correspondence categories might be called "analogies" and our analogy categories "relevant analogies") - truth is the minimal element of an analogy. The non-existence of a maximal element (making the semi-lattice into a lattice) corresponds to the possibility of forming different analogies between the same parts of a structure. One should not be tempted to call them "false" analogies because these may be ruled out by appropriate choice of the truth category. The possibility of two analogies not being encompassed by another (having no common upper bound, or even no upper bounds at all) corresponds to the possibility of two people having "different points of view" - you may form an analogy which helps you, and I may form a very different one that suits me, but providing they are both adequate for the task in hand (have the truth, at least, in common) the present theory does not attempt to judge between them - i.e. it leaves ample scope for debates on style, salubrious habits, and so on. If, however, these styles and habits become incorporated into the truth category then the theory does provide the necessary legalistic tools to enforce them. It is also able to comment that X's style implies Y's (i.e. forces X to do all that Y does plus some other mannerisms), or that Z's structured programming techniques encompass those of both X and Y.

Other useful concepts may be expressed in terms of analogy categories and Figs. 3 and 4. If we require the model to be an "emulator" then essentially we require it to reflect all the structure in the system emulated and the functor from the truth category to the modelled category becomes an isomorphism. The diagram of Fig. 3 then collapses to a triangle in which a faithful functor from the category to its model factors through the analogy. Milner [17] gives some interesting examples of "simulation" between programs within an algebraic framework that represents one concrete form of the abstract categories discussed here. The development of assertional programming languages may be seen as an attempt to make the model category isomorphic to the modelled category. The semi-lattice then becomes a lattice with the maximal element being isomorphic to them both. Fig. 1 may also be expanded with more model categories and we may consider analogy categories that are common to two or more models, i.e. the common features of different models. This sets up a further partial order on analogies that is compatible with that already defined and hence extends it.

Diagrams of possible relationships, such as those of Figs. 1 and 2, may now be seen as imbeddable in a whole web of analogy relations which express all the differing bases on which one may wish to compare the various structures. The rigour and practical utility of this web of relations is a function only of the extent to which we are prepared to define the items in the boxes in such diagrams - a not unexpected result! However, it is worth noting that virtually any attempts at formal definition are utilizable, from weak constraints to highly specific structures - the approach developed in this paper enables the mutual relationships implied by various definitions to be explored.

4. Conclusions

The concepts developed in this paper are global in nature rather than specific to particular aspects of the theory of computation or programming (technology or psychology). They do not conflict with or supersede the many current studies of the mathematical structure of programming itself, of virtual machines, of system analysis, or programmer psychology, and so on. Rather they provide tools for relating these diverse studies not only within their own frame of reference, but also globally in terms of the compatibility and conflict between prescriptions based on differing terms of reference and points of view. The term "structured programming" has come to mean a great many things to a great many people, and in its very diversity lies the danger that the momentum generated will be dissipated in a range of dogmas from different "schools". The formalism of "analogy categories" developed in this paper enables the essential cohesion of the various approaches to be expressed both rigorously and meaningfully on a basis of secure mathematical foundations.

5. References

1. Elspas, B., Levitt, K.N., Waldinger, R.J. and Waksmann, A., "An assessment of techniques for proving program correctness", ACM Comp. Surveys, Vol. 4, pp. 97-147, June 1972.
2. Foster, J.M. and Elcock, E.W., "Absys 1: an incremental compiler for assertions; an introduction", in Meltzer, B. and Michie, D., Machine Intelligence 4, pp. 423-429, Edinburgh: University Press 1969.
3. Chang, C.L. and Lee, C.T.L., Symbolic Logic and Mechanical Theorem Proving, New York: Academic Press 1973.
4. Arbib, M.A. and Manes, E.G., "Foundations of system theory", Automatica, Vol. 10, pp. 285-302, 1974.
5. Bobrow, L.S. and Arbib, M.A., Discrete mathematics, Ch. 9, Philadelphia: Saunders, 1974.
6. Holt, A.W., "Introduction to occurrence systems", in Jacks, E.L. (ed.) Associative Information Techniques, New York: Elsevier, 1968.
7. Scott, D., "The lattice of flow diagrams", in Dold, A. and Eckmann, B. (eds) Symposium on the semantics of algorithmic languages, pp. 311-366, Berlin: Springer, 1971.
8. Goguen, J.A., "Semantics of computation", in Proc. 1st Int. Symp. on Category Theory Applied to Computation and Control, Massachusetts, February 1974.
9. Dahl, O.J., Dijkstra, E.W. and Hoare, C.A.R., Structured Programming, New York: Academic Press, 1972.
10. Goldberg, R.P., "Survey of virtual machine research", Computer, Vol. 7, pp. 34-35, June 1974.
11. Popek, G.J. and Goldberg, R.P., "Formal requirements for virtualizable third generation architectures", Comm. ACM, Vol. 17, pp. 412-421, July 1974.
12. Gaines, B.R., "Varieties of computer - their applications and interrelationships", IFAC Symposium, Budapest, April 1968.
13. Goguen, J.A., "System theory concepts in computer science", Proc. 6th Hawaii Int. Conf. on System Sciences, pp. 77-80, 1973.
14. Goguen, J.A., "Systems and minimal realization", Proc. IEEE Conf. on Decision and Control, pp. 42-46, 1971.
15. Goguen, J.A., "Realization is universal", Math. Syst. Theory, Vol. 6, pp. 359-374, 1973.
16. Goguen, J.A., "Concept representation in natural and artificial languages: axioms, extensions, and applications for fuzzy sets", Int. J. Man-Machine Studies, Vol. 6, pp. 513-561, September 1974.
17. Milner, R., "An algebraic definition of simulation between programs", Proc. 2nd Int. Joint Conf. on Artificial Intelligence, London: British Computer Society, pp. 481-489, 1971.