The Varieties of Programming Language

C.A.R. Hoare
Oxford University Computing Laboratory

In 1973, Christopher Strachey wrote a monograph with the above title [8]. It began

> There are so many programming languages in existence that it is a hopeless task to attempt to learn them all. Moreover many programming languages are very badly described; in some cases the syntax, or rather *most* of the syntax, is clearly and concisely defined, but the vitally important question of the semantics is almost always dealt with inadequately.

Since that time, there has been a further spectacular increase in the variety of languages, and several of them have even entered common use. Completely new programming paradigms have emerged, based on logical inference, on functions and relations, or on the interaction of objects. New language features include communication, non-determinism, type inheritance, type inference, and lazy evaluation. Many programmers have become fluent in several languages; and many large applications have been constructed from parts implemented in different programming notations. Programming language designers are attempting to alleviate the resulting problems by combining the merits of several paradigms into a single more comprehensive language; but the signs of success in such a combination are difficult to recognise.

A solution to these problems may emerge from a wider understanding and agreement about the nature of programming and the choices available to the programming language designer and user. The current state of our understanding seems little better now than when Strachey wrote

> Not only is there no generally accepted notation, there is very little agreement even about the use of words. The trouble seems to be that programming language designers often have a rather parochial

> outlook and appear not to be aware of the range of semantic possibilities for programming languages. As a consequence they never explain explicitly some of the most important features of a programming language and the decisions among these, though rarely mentioned (and frequently I suspect made unconsciously), have a very important effect on the general flavour of the language...
> The unsatisfactory nature of our understanding of programming languages is shown up by the fact that although the subject is clearly a branch of mathematics, we still have virtually no theorems of general application and remarkably few specialised results.

This paper is dedicated to pursuit of the goals that Strachey set himself in his own work.

> The main purpose of this paper is to discuss some features of the range of semantic possibilities in programming language... A second purpose is to advocate a more conventionally mathematical approach to the problem of describing a programming language and defining its semantics, and, indeed, to the problems of computation generally.

To lighten the task, this paper concentrates on the following varieties of programming language.

1. Deterministic, like LISP or PASCAL: the possible result of executing each program is fully controllable by its environment or user.

2. Non-deterministic, like occam or Dijkstra's language [2]: the result of execution of a program may be affected by circumstances beyond the knowledge or control of its environment or user.

3. Strict, like LISP or Dijkstra's language: each stage of a calculation must be completed successfully before the start of the next stage which will use its results.

4. Lazy, like KRC or Miranda[1][9]: a calculation is started only when its result is found to be needed.

5. Interactive, like CSP or occam: the program communicates with its environment during execution.

6. Non-interactive, like the original FORTRAN: all the data required by the program may in principle be accumulated before the program starts; and all the results may be accumulated during the calculation for eventual output if and when the program terminates successfully.

[1]Miranda is a trade mark of Research Software Ltd.

Summary

The next section compares the merits of two methods of studying the mathematics of programming languages. The denotational method, due to Strachey, requires the construction of a mathematical meaning for each major component of each program expressed in each language. The various operators which combine the components are defined as functions, on the meanings of their operands. The algebraic approach avoids giving any meaning to the operators and operands; instead it formulates general equations describing the algebraic properties of the operators individually and in relation to each other. In some cases, this is sufficient to characterise the meaning of each operator, at least up to some form of equivalence or isomorphism. The algebraic approach seems to offer considerable advantages in the classification of the varieties of programming language, because it permits individual features and properties common to a range of languages to be characterised independently of each other.

The next important idea, due to Scott [7], is an ordering relation (approximation) between programs. This permits an elegant mathematical construction for the most essential of programming language features, namely recursion or iteration. Other benefits of the ordering include an elegant treatment of the phenomena of non-termination or even non-determinism. In software engineering, the same ordering can be used for stepwise development of specifications, designs, and programs.

For classifying the major varieties of programming language, the combination of the approximation ordering with an algebraic approach seems to offer the greatest benefits. Each variety is characterised by the different laws which they satisfy; and in many cases, the laws differ from each other only in the direction of the ordering between the two sides of an inequation. The final sections of this paper give several simple examples, relating to zero morphisms (**abort**), products (records, declarations), and coproducts (variants, cases).

The paper ends with a brief summary of related work, both completed and remaining to be started.

1 Denotational and Algebraic Approaches

A denotational semantics for a language defines the mathematical meaning of every syntactically valid program expressed in the language. A meaning is also given to every significant component of the program, in such a way that the meaning of the whole program is defined in terms of the meaning of its parts.

The study of denotational semantics has made a significant contribution to the development both of mathematics and of computing science. In the early days, Church's untyped lambda-calculus was selected as the branch of mathematics within which to formalise the meanings of programs. After the discovery of reflexive domains by Scott, a much more elegant and abstract basis for denotational semantics is found in domain theory.

In computing science, denotational semantics gives the programming language designer and theorist an excellent method of exploring the consequences of design decisions before committing the resources required to implement it, or the even greater resources required to test it in practical use. The semantics also provides a basis for the correct design and development of efficient implementations of the language, compatible across a variety of machines. It provides a conceptual framework for the user to specify programs before they are written, and to write them in a manner which reduces the danger of failing to meet that specification. And finally, it reveals most clearly the major structural variations between programming languages, even those that result from apparently quite minor decisions about individual features. When Dedekind, Frege and Russell gave interpretations within set theory of the long-familiar mathematical concept of number, they found similar major differences between the natural numbers, the integers, the fractions, the reals, and the complex numbers.

Because denotational semantics is so effective in revealing the differences between languages, it is correspondingly less effective in revealing their similarities. The essential similarities between the varieties of number are most clearly illuminated by listing the many algebraic properties which they share; the differences can then be isolated by listing the properties which they do not share. The dependence or independence of various selections of laws can be established by standard algebraic techniques. New kinds of mathematical structure can be discovered by combining algebraic laws in new ways. In some cases it is possible to prove that the meaning of each primitive concept and operation is defined uniquely by the laws which it obeys (perhaps up to some acceptable degree of isomorphism or equivalence). And finally, one may hope to classify all possible mathematical structures obeying a given basic set of reasonable laws.

All these properties of an algebraic approach are potentially beneficial to theorists of computing science. In view of the important role of algebra in the development, teaching and use of mathematics, it would be reasonable to expect similar benefits to the study of programming languages, their design, implementation, teaching and use. The algebraic approach is the one adopted in this paper for exploring the varieties of programming language.

The branch of algebra which seems to be most relevant for programming language theory is category theory. Its high level of abstraction avoids the clutter of syntactic detail associated with concrete programs, including declarations, variables, scopes, types, assignment and parameter passing. Its object structure neatly captures and takes advantage of the type structure of strictly typed languages, and the scope structure of languages with a concept of locality. And finally, it turns out that many of the features of a programming language are definable as categorical concepts in such a way that their effective uniqueness is guaranteed.

Exploration of the variety of programming languages requires consideration of categories which are enriched by an ordering relation [10]. These constitute a particularly simple kind of two-category, in which a preorder is given on each of its homsets. However, no knowledge of two-categories or even of category theory is required of the reader of this paper.

2 The Approximation Ordering

The approximation ordering between programs was originally introduced into programming language theory to provide an explanation of recursion. We shall use it as a general method of comparing the quality of any two programs written in the same language. If p and q are programs, $p \sqsubseteq q$ means that, for any purpose whatsoever and in any context of use, the program q will serve as well as program p, or even better. This is an ordering relation because it is reflexive and transitive

$$\begin{array}{ll} & p \sqsubseteq q \\ \text{if} & p \sqsubseteq q \text{ and } q \sqsubseteq r \text{ then } p \sqsubseteq r. \end{array}$$

We define two programs to be equivalent if they approximate each other

$$p \equiv q \;\;\widehat{=}\;\; p \sqsubseteq q \text{ and } q \sqsubseteq p.$$

Since we are willing to tolerate equivalent but non-identical programs, we do not require the approximation ordering to be antisymmetric.

This ordering relation may hold not only between programs but also between a specification and a program, or even between two specifications. In this case, $p \sqsubseteq q$ means that p and q are essentially the same specification, or that p is the more abstract or general specification and q is a more concrete or detailed specification, closer to the design of a program to meet the specification p. Every implementation of q is therefore an implementation of p, but not necessarily the other way round. In the extreme, q may itself be a program expressed in some efficiently executable programming language; this then is the final deliverable outcome of a development process that may have gone through several successive stages of more and more concrete design. By transitivity of $\sqsubseteq$, this final program meets the original specification with which the design process started. Thus the approximation ordering is as important for software engineering as it is for the theory of programming languages.

In a deterministic programming language the relation $p \sqsubseteq q$ holds if p always gives the same result as q, and q terminates successfully in all initial states from which p terminates, and maybe more; this is because non-termination is, for any purpose whatsoever, the least useful behaviour a program can display. In a non-deterministic language, an additional condition for $p \sqsubseteq q$ is that every possible outcome of executing q in a given environment should also be a possible outcome of executing p in the same environment; but maybe p is less deterministic, and might give some outcome that q will not. The program p is therefore less predictable, less controllable, or in short, a mere approximation to q. In a programming language like the typed lambda-calculus, all programs are terminating deterministic functions. One such program approximates another only if they compute the same function. As a result, the approximation ordering is an equivalence relation. The algebraic properties of such languages have been formulated directly in standard category theory; in this paper we will concentrate on languages with a non-trivial approximation ordering.

The $\sqsubseteq$ relation between programs is seen to have several different meanings, according to whether it is applied to a deterministic or a non-deterministic language. This is a characteristic feature of the algebraic method; for example, addition of complex numbers is very different from addition of integers; nevertheless the same symbol is used for similar purposes in both cases, for the very reason that the two meanings share their most important algebraic properties. The purpose of the $\sqsubseteq$ relation is to compare the quality of programs and specifications; and for this reason it must be transitive, and might as well be reflexive. Given that it has these properties, we are free to give the symbol the most appropriate interpretation for each individual language that we wish to study.

The *bottom* of an ordering relation (if it exists) is often denoted by $\perp$. It is defined (up to equivalence) by the single law

$$\perp \sqsubseteq q, \qquad \text{for all } q.$$

Proof. Let $\perp'$ be another bottom. Then $\perp \sqsubseteq \perp'$ because $\perp$ is a bottom; and $\perp' \sqsubseteq \perp$ because $\perp'$ is a bottom. Therefore $\perp \equiv \perp'$ □

The bottom program is for any purpose the most useless of all programs; for example, the program which never terminates under any circumstances whatsoever. In FORTRAN or machine code, the bottom is the tight loop

```
10 GO TO 10
```

In LISP it is the function *BOTTOM*, defined by a non-terminating recursion

```
(DEFFUN BOTTOM X    (BOTTOM X)).
```

In Dijkstra's language the bottom is called **abort**. This may fail to terminate; or being non-deterministic it may do even worse: it may terminate with the wrong result, or even the right one (sometimes, just to mislead you).

The *meet* of two elements p and q (if it exists) is denoted $p \sqcap q$. It is the best common approximation of both p and q. It is defined (up to equivalence) by the single law

$$r \sqsubseteq p \sqcap q \text{ iff } r \sqsubseteq p \text{ and } r \sqsubseteq q, \qquad \text{all } p, q, r.$$

Proof. Let $\sqcap'$ be another operator satisfying this law. By substituting $p \sqcap q$ for r,

$$\begin{aligned} p \sqcap q \sqsubseteq p \sqcap' q \quad & \text{iff} \quad p \sqcap q \sqsubseteq p \text{ and } p \sqcap q \sqsubseteq q \\ & \text{iff} \quad p \sqcap q \sqsubseteq p \sqcap q \end{aligned}$$

which is true by reflexivity of $\sqsubseteq$. The proof that $p \sqcap' q \sqsubseteq p \sqcap q$ is similar. □

Other algebraic properties of $\sqcap$ follow from the defining law. For example, it is associative, commutative, and idempotent, and

$$p \sqcap q \sqsubseteq p.$$

Proof. $p \sqcap q \sqsubseteq p \sqcap q$ implies $p \sqcap q \sqsubseteq p$ and $p \sqcap q \sqsubseteq q$ □

In a non-deterministic language, $p \sqcap q$ is the program which may behave like p or like q, the choice not being determined either by the programmer or by the environment of the program. In Dijkstra's language it would be written

```
**if** true → $p$
[] true → $q$
**fi**.
```

In this language, non-determinism is "demonic": if a program contains the non-deterministic possibility of going wrong, this is as bad as if it always goes wrong:

$$\bot \sqcap p \equiv \bot, \qquad \text{all } p.$$

If this law were not confirmed by ample experience of bugs in computer programs, it should still be adopted as a moral law by the engineer who undertakes to deliver reliable products.

Even a deterministic language has a meet. $(p \sqcap q)$ is the program that terminates and gives the same result as p and q when started from any initial state in which p and q both terminate *and both* give the *same* result. $(p \sqcap q)$ otherwise fails to terminate. More simply put, the graph of $p \sqcap q$ is the intersection of the graphs of p and q.

Although this description uniquely specifies the meaning of $(p \sqcap q)$, the operator $\sqcap$ is unlikely to be included explicitly in the notations of a deterministic programming language. Nevertheless it does exist, and can actually be programmed by

$$meet\ (p, q) \ \ \widehat{=}\ \ (\textbf{if}\ p = q\ \textbf{then}\ p\ \textbf{else}\ \bot).$$

A more efficient implementation of $(p \sqcap q)$ is simply to select an arbitrary member of the pair. By definition of $\sqcap$ this will be better than $(p \sqcap q)$, and an implementation should always be allowed to implement a program better (for all purposes) than the one written by the programmer. That is one of the motives for introducing the approximation ordering.

A program p is called *total* if it is a maximal element of the $\sqsubseteq$ ordering:

$$\text{if } p \sqsubseteq q \text{ then } q \sqsubseteq p, \qquad \text{for all } q.$$

A total program is one which is deterministic and always terminates. That is why it cannot be improved, either by extending the range of environments in which it terminates, or by reducing the range of its non-determinacy. Although we can reasonably point to a single worst program, unique up to equivalence, no reasonable programming language can have a single best program. If there were, it would be the best program for all purposes whatsoever, and there would be no point in using any other program of the language! Such a program would be a miracle (according to Dijkstra). But so far from solving all the problems of the world, its existence in programming language can only make that language futile.

The laws postulated so far (and those that can be deduced from them) are true in every programming language that admits the possibility of non-termination. It is not yet possible to distinguish varieties of language, not even to distinguish deterministic from non-deterministic languages. These and other important distinctions will be drawn in the next section.

3 Composition

Without much loss of generality, we can confine attention to languages in which there exists some method of composing programs p and q into some larger program called $(p; q)$. Execution of such a composite program usually (but not always) involves execution of both its components. In a procedural programming language like PASCAL or occam, we interpret this notation as sequential execution: q does not start until p has successfully terminated. In a functional language it denotes functional composition, such as might be defined in LISP

$$LAMBDA\ X(p(q\ X))).$$

Here is an example of an advantage of the algebraic approach: it ignores the spectacular difference in the syntax of composition in procedural and functional languages and concentrates attention on their essential mathematical similarities. Chief among the general properties of composition is associativity

$$p;(q;r) = (p;q);r.$$

Another property is the existence of a unit denoted I. It is uniquely defined by the equations

$$p;I = p = I;p, \qquad \text{for all } p.$$

Proof. Let I' be another unit. Because I' is a unit, $I = I';I$. Because I is a unit, $I';I = I'$. The result follows by transitivity of equality. □

In a procedural programming language, the unit is the trivial assignment $x := x$. In FORTRAN it is CONTINUE, in Dijkstra's language **skip**, and in LISP the identity function

$$(LAMBDA\ X\ X).$$

Another general property of composition is that it is monotonic, i.e., it respects the ordering of its operands:

$$\text{if } p \sqsubseteq q \text{ then } p;r \sqsubseteq q;r \text{ and } r;p \sqsubseteq r;q, \qquad \text{all } p, q, r.$$

If the preorder is regarded as giving a two-categorical structure to the programming language, monotonicity of composition is just a restatement of the interchange law. More generally, every operator of a programming language must be monotonic with respect to $\sqsubseteq$. Otherwise, there would be some context which would fail to be improved (or at least left the same) by replacing a worser component by a better. So any non-monotonic operator in a programming language would make the $\sqsubseteq$ relation useless for the purpose for which it is intended, namely to state that one program is better than another in all contexts of use. In algebra, that is one of the most convincing reasons for adopting a particular law as an axiom.

In a typed language, the composition of programs is undefined when the type of the result of the first component differs from that expected by the second component. This complication is elegantly treated in category theory by associating source and target objects with each arrow. Nevertheless, in this paper we will just ignore the complication, and assume that the source and target types of all the operands are the same. Where this is impossible, restrictions will be placed informally on the range of the variables.

It is well-known that the composition of two total functions is a total function. The same is true of total programs, which form a deterministic subset of a non-deterministic language. In fact all languages of interest will satisfy the law:

$$\text{if } p \text{ and } q \text{ are total, so is } (p; q), \qquad \text{all } p, q.$$

We have seen how the unit I of composition is uniquely defined. A zero of composition (if it exists) is denoted by Z. It is uniquely defined by the laws

$$p; Z = Z = Z; q, \qquad \text{for all } p \text{ and } q.$$

Proof. Let Z' also satisfy these laws. Then, because Z is a zero, $Z'; Z = Z$, and because Z' is a zero $Z' = Z'; Z$. The result follows by transitivity of equality. $\square$

In Dijkstra's programming language, the zero is the bottom program **abort**. To specify the execution of q after termination of **abort** cannot redeem the situation, because **abort** cannot be relied on to terminate. To specify execution of p before abortion is equally ineffective, because the non-termination will make any result of executing p inaccessible and unusable. In other words, composition in Dijkstra's language is *strict* in the sense that it gives bottom if either of its arguments is bottom.

However, in a language which interacts with its environment, the account given in the last paragraph does not apply. The program p may engage in some useful communications (perhaps even forever) thus postponing or avoiding the fate of abortion. So $(p; \bot)$ can be strictly better than $\bot$. The same is true in a language with jumps, since p can avoid the abortion by jumping over it. A similar situation obtains in a logic programming language like PROLOG, in which the ordering of the clauses of a program is significant. If a query can be answered by the rules in the first part of the program p, it does not matter if a later part aborts; but abortion at the beginning is an irrecoverable error.

In these languages, the bottom is a *quasi-zero* z, in a sense defined up to equivalence by the laws

$$z; p \sqsubseteq z \sqsubseteq q; z, \text{ all } p \text{ and } q.$$

Proof. let z' also be a quasi-zero. Then because z is a quasi-zero, $z \sqsubseteq z'; z$. Because z' is a quasi-zero $z'; z \sqsubseteq z'$. By transitivity of $\sqsubseteq$, $z \sqsubseteq z'$. A similar proof shows that $z' \sqsubseteq z$. The two quasi-zeroes are therefore equivalent. □

In a lazy functional programming language, the call of a function will not evaluate an argument unless the value of the argument is actually needed during evaluation of the body of the function. Such a mechanism of function call is said to be non-strict or lazy. Thus the function

$$DEFFUN\ K3 = LAMBDA\ X\ 3$$

can be successfully called by

$$(K3\ (\perp Y))$$

and will deliver the value 3, because no attempt is made to evaluate its argument $(\perp Y)$. However the call

$$\perp (K3\ Y)$$

will never succeed, no matter what the value of Y. This shows that $K3; \perp$ is actually worse than $\perp; K3$.

As a result, the bottom program in a lazy language is neither a zero nor a quasi-zero. In fact it is a quasi-cozero, in a sense defined up to equivalence by the laws

$$p; \perp \sqsubseteq \perp \sqsubseteq \perp; q, \qquad \text{for all } p \text{ and } q.$$

We have now explored the way in which composition interacts with the bottom program $\perp$. The question now arises how composition interacts with the $\sqcap$ operator. From the defining property of $\sqcap$ we can derive the following weak distribution law

$$r; (p \sqcap q); s \ \sqsubseteq\ (r; p; s) \sqcap (r; q; s).$$

Proof. $(p \sqcap q) \sqsubseteq p$ and $(p \sqcap q) \sqsubseteq q$. Because composition is monotonic, $r;(p \sqcap q);s \sqsubseteq r;p;s$ and $r;(p \sqsubseteq q);s \sqsubseteq r;q;s$. The law follows from the defining property of $\sqcap$. □

These laws hold in any programming language. However, in a truly non-deterministic language, the laws may be strengthened to an equation

$$r;(p \sqcap q);s = (r;p;s) \sqcap (r;q;s).$$

This strengthening is not valid in a functional or deterministic language.

The treatment of zeroes, quasi-zeroes and quasi-cozeroes is a simple example of the power of algebraic laws in classifying the varieties of programming language. Three clear varieties have emerged

(1) The strict non-interactive languages in which $\perp$ is a zero.

(2) The strict interactive languages in which $\perp$ is a quasi-zero.

(3) The non-strict languages in which $\perp$ is a quasi-cozero.

An orthogonal classification is that between deterministic languages and non-deterministic languages, in which composition distributes through $\sqcap$.

4 Products

If p and q are programs, we define their product $\langle p, q\rangle$ to be a program which makes a second copy of the current argument or machine state, and executes p on one of the two copies and q on the other one. The two results of p and q are delivered as a pair. This allows execution of p and q to proceed in parallel without interfering with each other. In a functional programming language with lists as a data structure, this can be defined:

$$\langle p, q\rangle \mathrel{\hat{=}} \lambda x.cons(px, qx).$$

In such a language, the duplication of the argument x is efficiently implemented by copying a machine address rather than the whole value of the argument. A procedural language has side-effects which update its initial machine state; so this implementation causes interference, and would not be valid. Instead, a completely fresh copy of the machine state is required; and this is what is provided by the *fork* feature of UNIX. The algebraic laws abstract from the radical differences of implementation, syntax, and general cultural environment between functional and parallel procedural programming paradigms. Some startling mathematical similarities are thereby revealed.

A stack-based language like PASCAL provides a restricted version of $\langle p, q\rangle$, where p is an expression delivering a reasonably small data value, and q represents the side-effect (if any) of evaluating the expression (usually, $q = I$). In such a language, the current machine state is a stack, and the effect of $\langle p, q\rangle$ is to pop the value of p onto the stack, making it available as the initial value of a new local variable to be accessed and updated by the body of the following block. By convention, we have put the top of the stack on the left.

If (x, y) is a pair of values, then we define the operation π to be one that selects the first of the pair (x), and μ selects the second of the pair (y). In LISP, these are the built-in functions *car* and *cdr* respectively. In a stack-based procedural language, μ has the effect of popping the stack, and is used for block exit. The operation π gives access to the most recently declared (innermost, local) variable. A combination of π and μ give access to variables declared in outer blocks; for example (1) $\mu; \pi$ gives the value of the next-to-local variable; (2) $\langle \mu; \pi, I\rangle$ makes a fresh copy of it as a new local variable; whereas (3) $\langle (\mu; \pi), \mu\rangle$ assigns it as a new value to the current most local variable. A high-level language uses identifiers to refer to these variables, and for practical programming this is a far better notation. However, in exploring the varieties of programming language, a notation closer to machine code seems paradoxically to be more abstract [1].

Each of the selectors π and μ would normally be expected to cancel the effect of a preceding operation of pairing, as described in the laws:

(1) $\langle p, q\rangle; \pi = p$

(2) $\langle p, q\rangle; \mu = q$

Furthermore, if $\langle \pi, \mu\rangle$ is applied to a pair, it will laboriously construct a pair from the first and second components of its argument, and thereby leave its argument unchanged. In general, if r is a program that produces a pair as result

(3) $\langle (r; \pi), (r; \mu)\rangle = r.$

These three laws are equivalent to the single biconditional law

$$(x; \pi = p \text{ and } x; \mu = q) \text{ iff } x = \langle p, q\rangle.$$

It is easy to calculate

$$\begin{array}{rcll} \langle \pi, \mu\rangle & = & \langle I; \pi, I; \mu\rangle \;=\; I & \text{(from (3))} \\ r; \langle p, q\rangle & = & \langle (r; \langle p, q\rangle; \pi), (r; \langle p, q\rangle; \mu)\rangle & \text{(from (3))} \\ & = & \langle (r; p), (r; q)\rangle & \text{(from (1),(2))} \end{array}$$

The laws given above define the concept of a product only up to isomorphism. Two programs p and q are said to be *isomorphic* if there exist an x and y such that

$$p; x = y; p,$$

where x and y are isomorphisms. An element x is said to be an *isomorphism* if there exists an $\breve{x}$ (known as the inverse of x) such that

$$x; \breve{x} = I = \breve{x}; x.$$

For example, I itself is an isomorphism ($I = \breve{I}$). Furthermore, if x and y are isomorphisms, so is $x; y (x; y; \breve{y}; \breve{x} = x; \breve{x} = I = \breve{y}; y = \breve{y}; \breve{x}; x; y)$. It follows that isomorphism between arrows is a reflexive relation ($p; I = I; p$), which is also transitive (if $p; x = y; q$ and $q; z = w; r$ then $p; x; z = y; q; z = y; w; r$), and symmetric (if $p; x = y; q$ then $\breve{y}; p; x; \breve{x} = \breve{y}; y; q; \breve{x}$, and hence $q; \breve{x} = \breve{y}; p$). In summary, isomorphism of programs is an equivalence relation.

If p is isomorphic to q ($p; x = y; q$), then the program p could equally well be implemented by the program q, merely preceding and following it by programs which are isomorphisms ($y; q; \breve{x}$). Similarly q could be implemented by means of p ($\breve{y}; p; x$). In either case, it would be impossible to distinguish the implementation from the original. That is why a collection of algebraic laws which defines a concept up to isomorphism should be accepted as an adequately strong definition in the theory of programming languages. It is certainly accepted as such in algebra or category theory.

To prove that products are defined up to isomorphism by laws (1) (2) and (3), we suppose that π', μ' and $\langle , \rangle'$ obey similar laws (1′), (2′) and (3′). We prove first that $\langle \pi, \mu \rangle'$ is the inverse of $\langle \pi', \mu' \rangle$:

$$\begin{array}{rcll}
\langle \pi, \mu \rangle'; \langle \pi', \mu' \rangle & = & \langle ((\langle \pi, \mu \rangle'; \pi'), (\langle \pi, \mu \rangle'; \pi') \rangle & \text{by } (3') \\
& = & \langle \pi, \mu \rangle & \text{by } (1'), (2') \\
& = & I. &
\end{array}$$

The proof that $\langle \pi', \mu' \rangle; \langle \pi, \mu \rangle' = I$ is similar. Now the required isomorphism is established by

$$\begin{array}{rcll}
\langle \pi', \mu' \rangle; \pi & = & \pi' & \text{by } (1) \\
\langle \pi', \mu' \rangle; \mu & = & \mu & \text{by } (2) \\
\langle p, q \rangle'; \langle \pi', \mu' \rangle & = & \langle ((\langle p, q \rangle'; \pi'), (\langle p, q \rangle'; \mu') \rangle & \text{by } (3) \\
& = & \langle p, q \rangle. & \text{by } (1'), (2').
\end{array}$$

The above definition of a product is standard in category theory. Unfortunately, it is not valid in a strict programming language, for which an implementation may insist on evaluating both components of a pair before proceeding with execution of the next instruction of the program. If evaluation of either component fails to terminate, the product also fails. Thus, for example

$$\langle p, \bot \rangle = \bot = \langle \bot, q \rangle.$$

A non-strict language avoids this problem, because its implementation does not evaluate either component of a pair until it is known to be needed [3]. If the very next operation is a selection (π or μ), this necessarily discards one of the components without evaluating it. So it does not matter if such evaluation would have failed.

The solution to this problem is to weaken the definition of a product to that of a *quasi-product*, in which some of the equalities are replaced by inequalities

(1) $\langle p, q \rangle; \pi \sqsubseteq p$

(2) $\langle p, q \rangle; \mu \sqsubseteq q$

(3) $r \sqsubseteq \langle (r; \pi), (r; \mu) \rangle$.

On the usual assumption of monotonicity, these three laws are equivalent to the single law

$$(x; \pi \sqsubseteq p \text{ and } x; \mu \sqsubseteq q) \ \text{ iff } \ x \sqsubseteq \langle p, q \rangle.$$

It is easy to calculate

$$I \sqsubseteq \langle \pi, \mu \rangle$$
$$r; \langle p, q \rangle \sqsubseteq \langle (r; p), (r; q) \rangle.$$

We also postulate that I, π and μ are total, and that the pairing function ($\langle\, ,\, \rangle$) maps total programs onto total programs. Under this hypothesis, we can show that the product is defined up to *quasi-isomorphism*, i.e., an isomorphism defined in terms of equivalence instead of equality.

Proof: very similar to the proof for products. Since $I, \pi, \mu, \pi', \mu', \langle \pi, \mu \rangle'$ and $\langle \pi', \mu' \rangle$ are all total, all inequations involving them are equivalences.

$$\begin{aligned} \langle \pi, \mu \rangle; \langle \pi', \mu' \rangle &\equiv \langle ((\langle \pi, \mu \rangle'; \pi'), (\langle \pi, \mu \rangle'; \pi') \rangle \\ &\equiv \langle \pi, \mu \rangle \equiv I. \end{aligned}$$
$$\langle \pi', \mu' \rangle; \pi \equiv \pi' \text{ and } \langle \pi', \mu' \rangle; \mu \equiv \mu'.$$

For the last clause we derive two inequations with similar proofs

$$\begin{aligned} \langle p,q\rangle';\langle \pi',\mu'\rangle &\sqsubseteq \langle p,q\rangle. \\ \langle p,q\rangle;\langle \pi,\mu\rangle' &\sqsubseteq \langle p,q\rangle. \end{aligned}$$

Apply "$;\langle \pi,\mu\rangle'$" to both sides of the first inequation to get

$$\langle p,q\rangle' \sqsubseteq \langle p,q\rangle;\langle \pi,\mu\rangle'$$

Together with the second inequation, this gives

$$\langle p,q\rangle;\langle \pi,\mu\rangle' \equiv \langle p,q\rangle'. \qquad \square$$

Unfortunately, in a non-deterministic programming language, the third law defining a quasi-product is not valid. Suppose r is a non-deterministic program, giving as answer either (3,3) or (4,4). Then $r;\pi$ can give answer 3 or 4, and so can $r;\mu$. Consequently, $\langle (r;\pi),(r;\mu)\rangle$ can give answers (3,3), (3,4), (4,3), or (4,4). This is certainly different from r, in fact more non-deterministic and therefore worse. In such a language, the inequality in the third law must be reversed

$$\langle (r;\pi),(r;\mu)\rangle \sqsubseteq r.$$

The consequences of such a reversal will not be explored here.

A similar analysis can be given for a coproduct. If p and q are programs, $[p,q]$ is a program whose execution involves execution of exactly one of p and q. The choice is made by testing some recognised "tag" component of its argument. In PASCAL this can be implemented by a **case** statement

case tag **of** 0 : p; 1 : q **end**.

Two operations are needed to put tags onto an argument: β puts on a 0 tag and γ puts on a 1 tag. As a result, we have two laws

(1) $\beta;[p,q] = p$

(2) $\gamma;[p,q] = q$.

These two laws are very similar to the first two laws defining the product, except that the order of the composition is reversed. The analogue of the third law is

(3) $[(\beta;r),(\gamma;r)] = r$.

In a strict language, this law is valid provided that r is an operation of a type that expects a tagged operand. $(\beta;r)$ selects the first alternative action of r, and $(\gamma;r)$ selects the second alternative. Putting these together again as a coproduct pair yields back the original r.

Unfortunately, this law is not valid in a non-strict programming language. In the case when evaluation of the tag field itself fails to terminate, the coproduct pair also fails to terminate. So if r is a non-strict function, which terminates even on a undefined argument, it is actually better than $[(\beta;r),(\gamma;r)]$. The third law must therefore be weakened to

$$[(\beta;r),(\gamma;r)] \sqsubseteq r.$$

Putting the three weaker laws together gives the biconditional law

$$(p \sqsubseteq \beta;x \text{ and } q \sqsubseteq \gamma;x) \text{ iff } [p,q] \sqsubseteq x.$$

A solution to these weakened equations is called a *quasi-coproduct.*

Discussion

This paper has suggested that many of the structural features of a programming language may be defined adequately by means of algebraic laws. The remaining features (basic types and primitive operations) can be given a conventional denotational semantics. The denotational and algebraic semantics can then be combined to give a complete semantics for the whole language. The combination is achieved by a familiar categorical construction: the language is defined as the free object over the denotational semantics in the variety of algebras which obey the algebraic equations. The construction needs slight adaptation to deal with inequations instead of equations.

In a language with strong typing, the operations of the language (both primitive and non-primitive) are defined only in contexts which satisfy the constraints of the type system. A type system can be conveniently represented in the object structure of a category. The free construction which defines the semantics of the language now yields a heterogeneous algebra, with a sort corresponding to each homset. If the language allows the programmer to define new types, the set of objects themselves have to be defined as a free (word) algebra over the primitive types of the language.

The advantage of the two-level denotational and algebraic semantics of a programming language is the potential simplification that results from separate treatments of the different features of the language. Another potential advantage is that the algebraic laws can be used directly in the optimisation of programs by means of correctness-preserving transformations. But perhaps the main advantage is that it permits the denotational semantics of the primitive types and operations of the language to be varied, while the algebraic semantics of the more structured features of the language are held constant. Thus each implementation of the language defines a different operational semantics, which can be proved correct with respect to the same denotational semantics. This proof is achieved by means of an abstraction function (or "retrieve" function of VDM), which plays the role of a natural transformation between functors in category theory.

It is hoped that further exploration of these topics will appear in the literature [4, 5, 6].

Acknowledgements.

The author is grateful to the Admiral R. Inman Chair of Computing Science at the University of Texas at Austin for providing an opportunity for an initial study of Category Theory. Also to He Jifeng for technical assistance with the mathematical basis of this article.

References

[1] G. Cousineau, P.L. Curien, M. Mauny. The Categorial Abstract Machine in Functional Languages and Computer Architecture. LNM 201 Springer-Verlag.

[2] E.W. Dijkstra. A Discipline of Programming. Prentice Hall.(1976).

[3] D.P. Friedman and D.S. Wise. Aspects of Applicative Programming for Parallel Processing. IEEE Trans. Comp. C-27 (1978) 289-296.

[4] C.A.R. Hoare and He, Jifeng. Natural Transformations and Data Refinement. Information Processing Letters (to appear).

[5] C.A.R. Hoare and He, Jifeng. Two-categorical Semantics for Programming Languages. (In preparation).

[6] C.A.R. Hoare and He, Jifeng. Data Refinement in a Categorical Setting. (In preparation).

[7] D.S. Scott. Data Types as Lattices. SIAM Journal of Computing 5 (1976) 552-587.

[8] C. Strachey. The varieties of Programming Language. PRG-10. (1973).

[9] D.A. Turner. Miranda, a non-strict Functional Language with Polymorphic Types. Lecture Notes in Computer Science 201 (1885) 1-16.

[10] M Wand. Fixed-Point Constructions in Order-Enriched Categories. THEOR. COM. 8(1) 13-30 (1979).