LAWS OF PROGRAMMING

A complete set of algebraic laws is given for Dijkstra's nondeterministic sequential programming language. Iteration and recursion are explained in terms of Scott's domain theory as fixed points of continuous functionals. A calculus analogous to weakest preconditions is suggested as an aid to deriving programs from their specifications.

C. A. R. HOARE, I. J. HAYES, HE JIFENG, C. C. MORGAN, A. W. ROSCOE, J. W. SANDERS, I. H. SORENSEN, J. M. SPIVEY, and B. A. SUFRIN

Some computer scientists have abandoned the search for rational laws to govern conventional procedural programming. They tend to recommend the use of functional programming [2] or logic programming [10] as alternatives. Here we shall substantiate a claim that conventional procedural programs are mathematical expressions, and that they are subject to a set of laws as rich and elegant as those of any other branch of mathematics, engineering, or natural science.

Furthermore, we suggest that a comprehensive set of algebraic laws serves as a useful formal definition (axiomatization) of a set of related mathematical notations, and specifically of a programming language—a suggestion due originally to Igarishi [8]. The algebraic laws provide an interface between the user of the language and the mathematical engineer who designs it. Of course, the mathematician should also design a model of the language, to check completeness and consistency of the laws, to provide a framework for the specifications of programs, and for proofs of correctness.

Here are some of the familiar laws of arithmetic, which apply to multiplication of real numbers:

(1) Multiplication is *symmetric*, or in symbols,

$$x \times y = y \times x, \quad \text{for all numbers } x \text{ and } y.$$

It is conventional in quoting laws to omit the phrase "for all x and y in the relevant set."

(2) Multiplication is *associative*, or in symbols,

$$x \times (y \times z) = (x \times y) \times z.$$

It is conventional to omit brackets for associative operators and write simply $x \times y \times z$.

(3) Multiplication by zero always gives zero:

$$0 \times x = 0.$$

Zero is said to be a *fixed point* or *zero* of multiplication.

(4) Multiplication by one leaves a number unchanged:

$$1 \times x = x.$$

The number one is said to be an *identity* or a *unit* for multiplication.

(5) Division is the *inverse* of multiplication:

$$y \times (x/y) = x, \quad \text{provided} \quad y \neq 0.$$

Another law relating multiplication and division is

$$z/(x \times y) = (z/x)/y, \quad \text{provided} \quad y \neq 0 \text{ and } x \neq 0.$$

(6) Multiplication *distributes* through addition:

$$(x + y) \times z = (x \times z) + (y \times z).$$

It is usual for brackets to be omitted on the right-hand side of this equation, on the convention that a distributive operator binds tighter than the operator through which it distributes.

(7) Multiplication by a nonnegative number is monotonic, in the sense that it preserves ordering in its other operand, or in symbols,

$$x \le y \Rightarrow x \times z \le y \times z, \qquad \text{provided} \quad z \ge 0.$$

If either factor is reduced, the value of the product does not increase.

(8) Multiplication is *continuous* in the sense that it preserves the limit of any convergent sequence of numbers:

$$(\lim_{n\to\infty} x_n) \times y = \lim_{n\to\infty}(x_n \times y), \qquad \text{provided} \quad x_n \text{ converges.}$$

(9) If we define

$$x \cap y = \text{the lesser of } x \text{ and } y$$

$$x \cup y = \text{the greater of } x \text{ and } y,$$

then we have the following laws:

$$x \cap y = y \cap x$$

$$(x \cap y) \ge z \equiv x \ge z \wedge y \ge z$$

$$(x \cup y) \le z \equiv x \le z \wedge y \le z$$

$$x \cap (y \cup z) \equiv (x \cap y) \cup (x \cap z).$$

The mathematician or engineer will be intimately familiar with all these laws, having used them frequently and intuitively. The applied mathematician, scientist, or engineer will also be familiar with many relevant laws of nature and will use them explicitly to find solutions for otherwise intractable problems. Ignorance of such laws would be regarded as a disqualification from professional practice. What then are the laws of programming, which provide the formal basis for the profession of software engineering?

Many programmers may be unable to quote even a single law. Many who have suffered the consequences of unreliable programs may claim that programmers do not observe any laws. This accusation is both unfair and inappropriate. The laws of programming are like the laws of arithmetic. They describe the properties of programs expressible in a suitable notation, just as the laws of arithmetic describe the properties of numbers, for example, in decimal notation. It is the responsibility of programming language designers and implementors to ensure that programs obey the appropriate collection of useful, elegant, and clearly stated laws.

The designers of computer hardware have a similar responsibility to ensure that their arithmetic units obey laws such as the monotonicity of multiplication (7). Regrettably, several computer designs have failed to do so. Similarly, many current programming languages fail to obey even the most obvious laws such as those expounded in this paper. Occam [11] is one of the first languages to be deliberately designed to obey such mathematical laws. The language used in this paper is simpler than Occam, in that it omits communication and concurrency. Laws that are not valid in the more complex language will be noted.

THE LANGUAGE

In order to formulate mathematical laws, it is necessary to introduce some notation for describing programs. We shall use a notation (programming language) that is especially concise and suitable for its purpose, based on the language introduced in [4]. It has three kinds of primitive command and five methods of composing larger commands (programs). The *SKIP* command is denoted II. Execution of this command terminates successfully, leaving everything unchanged.

The *ABORT* command is denoted $\perp$. It places no constraint on the behavior or misbehavior of the executing machine, which may do anything, or fail to do anything; in particular, it may fail to terminate. Thus $\perp$ represents the behavior of a broken machine, or a program that has run wild. The most important property of *ABORT* is that it is a program that no one would ever want to use or write. It is like a singularity in a mathematical system that must, at all costs, be avoided by the competent engineer. In order to prove the absence of such an error, one must use a mathematical theory that admits its existence.

In the *Assignment* command, let x be a list of distinct variables, and let E be a list of the same number of expressions. The assignment $x := E$ is executed by evaluating all the expressions of E (with all variables taking their most recently assigned values) and then assigning the value of each expression to the variable at the same position in the list x. This is known as multiple or simultaneous assignment. We assume that expressions are evaluated without side effects and stipulate that the values of the variables in the list x do not change until all the evaluations are complete. For simplicity, we shall also assume that all operators in all expressions are defined for all values of their arguments, so that the evaluation of an expression always terminates successfully. This assumption will be relaxed in the section on undefined expressions.

Sequential composition. If P and Q are programs, (P; Q) is a program that is executed by first executing P. If P does not terminate, neither does (P; Q). If and when P terminates, Q is started; and then (P; Q) terminates when Q does.

Conditional. If P and Q are programs and b is a Boolean expression, then (P $\lhd$b$\rhd$ Q) is a program. It is executed by first evaluating b. If b is true, then P is executed, but if b is false, then Q is executed instead. The more usual notation for a conditional is **if** b **then** P **else** Q. We have chosen an infix notation $\lhd$b$\rhd$ because it simplifies expression of the relevant algebraic laws.

Nondeterminism. If P and Q are programs, then (P $\cup$ Q) is a program that is executed by executing either P or Q. The choice between them is arbitrary. The programmer has deliberately postponed the decision, possibly to a later stage in the development of the program, or possibly has even delegated the decision to the machine that executes the program.

Iteration. If P is a program and b is a Boolean expression, then (b * P) is a program. It is executed by first evaluating b; if b is false, execution terminates successfully, and nothing is changed. But, if b is true, the machine proceeds to execute P; (b * P). A more conventional notation for iteration is **while** b **do** P.

Recursion. Let X be the name of a program that we will define by recursion, and let $F(X)$ (containing occurrences of the name X) be a program text defining its behavior. Then $\mu X.F(X)$ is a program that behaves like $F(\mu X.F(X))$; that is, all recursive occurrences of the program name have been replaced by the whole recursive program. This fact is formalized in the following law:

$$\mu X.F(X) = F(\mu X.F(X)).$$

In mathematics, substitution of equals is always allowed and may be repeated indefinitely:

$$\mu X.F(X) = F(\mu X.F(X)) = F(F(\mu X.F(X))) = \ldots.$$

This is essentially the way that recursively defined programs are executed by computer. Of course, iteration is only a special case of recursion:

$$b * P = \mu X.(P;\ X) \lhd b \rhd \Pi.$$

Iteration is simpler and more familiar than general recursion, and so is worth treating separately. An understanding of or liking for recursion is not needed for appreciation of this article.

As an example of the use of these notations, here is a program to compute the quotient q and remainder r of division of nonnegative x by positive y. It offers a choice of methods, one of which terminates when $y = 0$.

$$(q,\ r := 0,\ x;\ (r \geq y * (q,\ r := q + 1,\ r - y)))$$

$$\cup\ ((q,\ r := x \div y,\ x \text{ rem } y) \lhd y \neq 0 \rhd q := 0).$$

The free use of brackets around the subexpressions of a computer program may seem unfamiliar, but follows naturally from our decision to treat programs as mathematical formulas. Sometimes it is convenient to omit brackets, provided they can be reinserted by precedence rules such as the following:

, binds tightest

:=

*

;

$\lhd b \rhd$

$\cup$ binds loosest.

Normal arithmetic operators bind tightest of all. Thus the example quoted above could have been written without any brackets.

The notations of our language can be defined in terms of Dijkstra's language of guarded commands:

$$P \cup Q = \textbf{if } \text{true} \rightarrow P \,\square\, \text{true} \rightarrow Q \textbf{ fi}$$

$$P \lhd b \rhd Q = \textbf{if } b \rightarrow P \,\square\, \neg b \rightarrow Q \textbf{ fi},$$

where $\neg b$ is the negation of b

$$b * P = \textbf{do } b \rightarrow P \textbf{ od}.$$

Conversely, guarded commands can be defined in terms of the notations given above, for example:

$$\textbf{if } b \rightarrow P \,\square\, c \rightarrow Q \textbf{ fi} = ((P \cup Q) \lhd c \rhd P) \lhd b \rhd (Q \lhd c \rhd \bot)$$

$$\textbf{do } b \rightarrow P \,\square\, c \rightarrow Q \textbf{ od} = (b \vee c) * (\textbf{if } b \rightarrow P \,\square\, c \rightarrow Q \textbf{ fi}).$$

Thus any program expressed in Dijkstra's language can be translated into our language, but this may cause an explosion in the length of the code for guarded command sets. Any program expressed in our notation, but restricted to iteration as the only form of recursion, can be translated into Dijkstra's language. We make no claim of notational superiority, except that our notation permits more succinct expression of some of the laws.

This description of the commands of our programming language is quite informal and deliberately fails to give a mathematical definition of the concept of a program. Experienced programmers will understand our intention. It would be inappropriate to postpone a study of the laws of arithmetic until after giving the traditional foundationary definitions (e.g., that a cardinal is a class of sets related by one-one functions). Indeed, even a mathematician gains a clearer, deeper, and more useful understanding of a concept by the study of its properties rather than its formal definition. One of the objectives of this article is to propose that the details of the design of a programming language can also be neatly explained by algebraic laws.

Technical Notes

It is usual in theoretical texts to make a sharp distinction between concrete notations like numerals (expressed in a variety of bases) and the abstract objects for which they stand (e.g., the natural numbers). In this article the word *program* will stand for an abstract concept, roughly equated with *the range of possible observable effects of executing its text in a variety of initial states.* When we wish to emphasize the specific concrete text being manipulated by program transformation, we will call it a *program text.* But, in general, we shall take a relaxed attitude toward these formal distinctions, as do most applied mathematicians and engineers.

The laws given in this article can be regarded as a completely formal algebraic specification of our programming language. The laws are strong enough to permit each program, not involving recursion, to be reduced to a normal form. A natural partial ordering is defined for normal forms, and programs can be identified with ideals in this partial ordering. The ideal construction is somewhat reminiscent of the Dedekind definition of a real number as a certain set of rationals.

Among theoreticians, it is common to identify a non-

deterministic program with a relation, namely, the set of all pairs of states of a machine such that if the program starts in the first state of the pair then it can end in the second state. For example, Il would be the identity relation, and *ABORT* would be the universal relation. However, this definition would not be correct for our language, since it would invalidate several of the laws (e.g., see (3) under "Sequential Composition" and (4) under "Limits"). In order to make these laws true, it is necessary to define programs as a particular *subset* of relations that have special properties. For example,

a program is a *total* relation;
the image of each state is finite or universal.

Nontermination has to be represented by a fictitious "state at infinity" that can be "reached" only by a nonterminating program. Also, if the fictitious state is in the image of a state, then that image is universal. These properties are preserved by all the operators of our language. For further details see [7].

The fictitious state involves both controversy and complexity, both of which are irrelevant to the needs of practicing programmers. For practicing engineers, mathematical laws are also more relevant than the elaborate models constructed in a study of foundations.

Another problem with models is that they change quite radically when the programming language is extended, for example, by introduction of input and output commands. Such extensions can be (and perhaps should be) designed to preserve the validity of most of the laws obeyed by the simpler language, just as the arithmetic of real numbers shares many of the properties of integer arithmetic. This has been done for Occam in [11].

Summary

The laws presented here not only apply to concrete programs, as expressed in the notations of the programming language described in the first section, but most also apply to program specifications, expressible in a wider range of more powerful notations. Additional laws are given to assist in the stepwise development of designs from specifications and programs from designs. In fact, we shall study a series of four classes of object, where each class includes its predecessor in the series and obeys all or almost all of the same laws.

(1) Finite programs are expressible in the notations of the programming language, excluding iteration and recursion. Laws for finite programs are given under "Algebraic Laws." They are sufficiently powerful to permit every finite program text to be reduced to a simple normal form.

(2) Concrete programs are expressible in the full programming language, including recursion.

(3) Abstract programs are expressible by means of programming notations plus an additional operator for denoting a limit of a convergent sequence of consistent programs. The relevant concepts and laws are those of domain theory and are explained under "Domain Properties."

Objects in the first three classes are called programs, and they all satisfy the laws of programming explained under "Algebraic Laws" and "Domain Properties."

(4) The remaining class is that of specifications. This is the most general class because there is no restriction on the notations in which they may be expressed. Any well-defined operator of mathematics or logic may be freely used, even including negation. The laws that apply to specifications are useful in the stepwise development of designs and programs to meet their specifications. The price of the greater notational freedom of expression of specifications is that it is possible (and easy) to write specifications that cannot be satisfied by any program. Another source of potential difficulty is that specifications fail to obey some of the laws that are valid for programs.

The distinction between these classes may seem complicated, but is actually as simple as the familiar distinctions made between different classes of numbers.

(1) Finite programs can be compared to rational numbers. Algebraic laws permit all arithmetic expressions to be reduced to a ratio of coprime integers, whose equality can be easily established.

(2) Concrete programs are like algebraic real numbers, which are definable within a restricted notational framework (as solutions of polynomial equations). They constitute a denumerable set.

(3) Abstract programs are like real numbers; they enjoy the property that convergent sequences have a limit. For many purposes (e.g., calculus) real numbers are far more convenient to reason with than algebraic numbers. They form a nondenumerable set.

(4) Specifications may be compared to complex numbers, where more operators (e.g., square root) are total functions. The acceptance of imaginary numbers may be difficult at first, because they cannot be represented in the one-dimensional real continuum. Furthermore, they fail to obey such familiar laws as $x^2 \geq 0$. Nevertheless, it pays to use them in definition, calculation, and proof, even for problems where the eventual answers must be real. In the same way, specifications are useful (even necessary) in requirements analysis and program development, even though they will never be executed by computer.

It might seem preferable to report a case study in which the laws had been used to assist in the development of a correct program of substantial size. Unfortunately, this is not possible. The task of writing a substantial program requires more application-orientated mathematics than the elementary algebra presented in this article. One would not expect to illustrate the laws of arithmetic by a case study in the design of a bridge. The laws of programming are broad and shallow, like the laws of arithmetic. They should be learned and used intuitively, like the grammatical rules of a foreign language.

ALGEBRAIC LAWS

About 30 algebraic laws obeyed by finite programs, that is, programs expressible without iteration or recursion, will be presented in this section. The laws are sufficiently powerful to permit every finite program to be reduced to a simple normal form, which can be used to test whether any two such texts denote the same program.

We shall adopt the following conventions for the range of variables:

P, Q, R	stand for programs.
b, c, d	stand for Boolean expressions.
e, f, g	stand for single expressions.
E, F, G	stand for lists of expressions.
x, y, z	stand for lists of assignable program variables, where no variable appears more than once in the combined list x, y, z.

Furthermore, x is the same length as E, y the same length as F, and z the same length as G.

Technical Notes

(1) The phrase "P stands for a program" may be ambiguous. Does P stand for some mathematical object that can be expressed in several different ways in a certain programming notation? Or does P stand for the *text* of a program, which must replace P before any law containing P is used? A reasonable answer to these questions is that it does not matter. Consider the analogy of the mathematical equation

$$n + m = m + n.$$

The variables n and m are normally considered to represent actual numbers, independent of the way in which they are expressed. But the law is equally valid if n and m are replaced by numerals, that is, sequences of digits in ternary notation, for example. Furthermore, it is equally valid if n is consistently replaced by any other arithmetic expression, such as $2 \times p^2 + q$ in $2 \times p^2 + q + m = m + 2 \times p^2 + q$.

(2) Similar questions may be asked about the meaning of the equations that embody the algebraic laws of programming and that assert the identity of two programs written in different ways. Clearly it cannot be the *texts* of the programs that are stated to be equal, but rather the *meanings*. The meaning of a program can be roughly understood as the possible effects of its execution in each initial state of the computer. For example, the following equations are true:

$$(x := 037) = (x := 37)$$

$$(x := y \times y + 2 \times y \times x + z \times z) = (x := (y + z) \times (y + z))$$

$$(x := 0; x := 1) = (x := 1).$$

In each case, the two programs are equal, even though one of them may be slower in execution than the other. This reflects a deliberate decision to abstract from questions of execution speed, with the explicit objective of allowing inefficient program texts to be replaced by more efficient representations of the same program. Thus the laws of programming can be used as correctness-preserving transformations, or as justification for automatic code optimization. That is the practical reward for designing, implementing, and using languages with mathematical properties.

(3) On several occasions we shall introduce new notations into the laws, which are not included in the programming language, but that describe programs that could be so expressed. This is a fairly familiar mathematical practice. For example, a polynomial in x can be defined syntactically as a text that is either a constant or a polynomial multiplied by x and added to a constant. Thus the following are polynomials:

$$0,\ 7,\ 7 \times x + 4,\ (7 \times x + 4) \times x + 17,\ \ldots.$$

However, a polynomial is often more conveniently described using summation ($\sum$) and exponentiation,

$$\sum_{i \le n} a_i \times x^i,$$

even though these notations are not included in the formal language of polynomials.

Nondeterminism

The laws governing nondeterministic choice apply to all kinds of choice. The equations given below assert the identity of the whole range of choices described by their left- and right-hand sides. Of course, a particular selection made from the left-hand side may differ from a particular selection made from the right-hand side. This is also true of two selections made from the *same* text. So, in the presence of nondeterminism, two executions of the same program test do not necessarily give the same result.

(1) Clearly, it does not make any difference in what order a choice is offered. "Tea or coffee?" is the same as "coffee or tea?."

$$P \cup Q = Q \cup P \qquad \text{(symmetry)}$$

(2) A choice between three alternatives (tea, coffee, or cocoa) can be offered as first a choice between one alternative and the other two, followed (if necessary) by a choice between the other two, and it does not matter in which way the choices are grouped.

$$P \cup (Q \cup R) = (P \cup Q) \cup R \qquad \text{(associativity)}$$

(3) A choice between one thing and itself offers no choice at all (Hobson's choice).

$$P \cup P = P \qquad \text{(idempotence)}$$

(4) The *ABORT* command already allows completely arbitrary behavior, so an offer of further choice makes no difference to it.

$$\bot \cup P = \bot \qquad \text{(zero } \bot\text{)}$$

This law is sometimes known as Murphy's Law, which states, "If it can go wrong, it will"; the left-hand side describes a machine that *can* go wrong (or can behave like P), whereas the right-hand side might be taken to describe a machine that *will* go wrong. But the true meaning of the law is actually worse than this: The program ⊥ will not always go wrong—only when it is most disastrous for it to do so! The abundance of empirical evidence for law (4) suggests that it should be taken as the first law of computer programming.

A choice between $n + 1$ alternatives can be expressed more briefly by the indexed notation

$$\bigcup_{i \le n} P_i = P_0 \cup P_1 \cup \cdots \cup P_n.$$

The indexed notation need not be included in the programming language. Nevertheless, it is useful in formulating the laws, since it enables a single law to be applied to an arbitrary number of alternatives. In each application of the laws to an actual program text, the list of alternatives would be written in full.

The Conditional

For each given Boolean expression b, the choice operator ◁b▷ specifies a choice between two alternatives with one written on each side. The first two laws clearly express the criteria for making this choice, that is, the truth or falsity of b.

(1) $P \lhd true \rhd Q = P.$

(2) $P \lhd false \rhd Q = Q.$

Like ∪, the conditional is idempotent and associative:

(3) $P \lhd b \rhd P = P.$

(4) $P \lhd b \rhd (Q \lhd b \rhd R) = (P \lhd b \rhd Q) \lhd b \rhd R.$

Furthermore, it satisfies the less familiar laws

(5) $P \lhd b \rhd Q = Q \lhd \neg b \rhd P,$

where ¬b is the negation of b;

(6) $P \lhd c \lhd b \rhd d \rhd Q = (P \lhd c \rhd Q) \lhd b \rhd (P \lhd d \rhd Q),$

where c ◁b▷ d is a conditional expression, giving value c if b is true and d if b is false; and

(7) $P \lhd b \rhd (Q \lhd b \rhd R) = P \lhd b \rhd R.$

These laws may be checked by considering the two cases when b is true and when it is false. For example, law (7) states that the middle operand Q is not selected in either case.

Suppose one of the operands of a conditional offers a nondeterministic choice between P and Q. Then it does not matter whether this choice is made before evaluation of the condition or afterward, since the value of the condition is not affected by the choice:

(8) $(P \cup Q) \lhd b \rhd R = (P \lhd b \rhd R) \cup (Q \lhd b \rhd R).$

From this can be deduced a similar law for the right operand of ◁b▷:

(9) $R \lhd b \rhd (P \cup Q) = (R \lhd b \rhd P) \cup (R \lhd b \rhd Q).$

PROOF.

$$LHS = (P \cup Q) \lhd \neg b \rhd R$$
$$= (P \lhd \neg b \rhd R) \cup (Q \lhd \neg b \rhd R) = RHS \quad \text{(by (5))}. \ \square$$

An operator that distributes like this through ∪ is said to be *disjunctive*.

Any operation that does not change the value of the Boolean expression b will distribute through ◁b▷. An example is nondeterministic choice. It does not matter whether the choice is exercised before or after evaluation of b:

(10) $(P \lhd b \rhd Q) \cup R = (P \cup R) \lhd b \rhd (Q \cup R).$

For the same reason, a conditional ◁c▷ distributes through another conditional with a possibly different condition ◁b▷:

(11) $(P \lhd b \rhd Q) \lhd c \rhd R = (P \lhd c \rhd R) \lhd b \rhd (Q \lhd c \rhd R).$

Using these laws we can prove the theorem

(12) $(P \lhd c \rhd R) \lhd b \rhd (Q \lhd d \rhd R)$
$= (P \lhd b \rhd Q) \lhd c \lhd b \rhd d \rhd R.$

PROOF.

$$RHS = ((P \lhd b \rhd Q) \lhd c \rhd R) \lhd b \rhd ((P \lhd b \rhd Q) \lhd d \rhd R) \quad \text{(by (6))}$$
$$= ((P \lhd c \rhd R) \lhd b \rhd (Q \lhd c \rhd R)) \lhd b \rhd ((P \lhd d \rhd R) \lhd b \rhd (Q \lhd d \rhd R)) \quad \text{(by (11))}$$
$$= LHS \quad \text{(by (7) and (4))}. \ \square$$

Sequential Composition

(1) Sequential composition is associative; to perform three actions in order, one can either perform the first action followed by the other two or the first two actions followed by the third.

$$P; (Q; R) = (P; Q); R \quad \text{(associativity)}$$

(2) To precede or follow a program P by the command II (which changes nothing) does not change the effect of the program P.

$$(II; P) = (P; II) = P \quad \text{(unit II)}$$

(3) To precede or follow a program P by the command ⊥ (which may do anything whatsoever) results in a program that may do anything whatsoever—it may even behave like P!

$$(\bot; P) = (P; \bot) = \bot \quad \text{(zero } \bot)$$

The law $P; \bot = \bot$ states that we are not able to observe anything that P does before P; ⊥ reaches ⊥. This law will not be true for a language in which P can interact with its environment, for example, by input and output.

The informal explanation of this law is weak. Perhaps it is better explained as a moral law. The program $\perp$ is one that the programmer has a duty to avoid. Equally, the sequential programmer has the duty to avoid both ($\perp$; P) and (P; $\perp$). It is pointless to draw distinctions between programs that must be avoided anyway.

(4) A machine that selects between P and Q, and then performs R when the selected alternative terminates, cannot be distinguished from one that initially selects whether to perform P followed by R or Q followed by R:

$$(P \cup Q); R = (P; R) \cup (Q; R).$$

For the same reason, composition distributes rightward through $\cup$:

$$R; (P \cup Q) = (R; P) \cup (R; Q).$$

In summary, sequential composition is a disjunctive operator.

(5) Evaluation of a condition is not affected by what happens afterwards, and therefore ; distributes leftward through a conditional:

$$(P \lhd b \rhd Q); R = (P; R) \lhd b \rhd (Q; R).$$

However, ; does not distribute rightward through a conditional, so in general it is not true that R; (P $\lhd$b$\rhd$ Q) = (R; P) $\lhd$b$\rhd$ (R; Q). On the left-hand side, b is evaluated after executing R, whereas on the right-hand side it is evaluated before R; and in general, prior execution of R can change the value of b.

Assignment

It is a law of mathematics that the value of an expression is unchanged when the variables it contains are replaced by expressions or constants denoting the values of each variable. If E(x) is a list of expressions and F is a list of the values of the variables x, then E(F) is a copy of E in which every occurrence of each variable of x is replaced by a copy of the expression occupying the same position in the list F.

(1) This convention is used in the first law of assignment, which permits merging of two successive assignments to the same variables:

$$(x := E; x := F(x)) = (x := F(E)).$$

For example,

$$(x := x - 1; x := 2 \times x + 1)$$
$$= (x := 2 \times (x - 1) + 1).$$

(2) The second law states that the assignment of the value of a variable back to itself does not change anything:

$$(x := x) = \text{II}.$$

This law is false for a language in which access to an uninitialized variable leads to a different effect, for example, abortion.

(3) In fact, such a vacuous assignment can be added to any other assignment without changing its effect (recall that x and y are disjoint):

$$(x, y := E, y) = (x := E).$$

(4) Finally, the lists of variables and expressions may be subjected to the same permutation without changing the effect of the assignment:

$$(x, y, z := E, F, G) = (y, x, z := F, E, G).$$

Corollary. $(x, y := E, F) = (y, x := F, E)$.

These four laws together are sufficient to reduce any sequence of assignments to a single assignment. For example,

$$(x, y := F, G; y, z := H(x, y), J(x, y))$$
$$= (x, y, z := F, G, z; x, y, z := x, H(x, y), J(x, y)) \quad \text{(by (3) and (4))}$$
$$= (x, y, z := F, H(F, G), J(F, G)) \quad \text{(by (1))}.$$

(5) Assignment distributes rightward through a conditional, changing occurrences of the assigned variables in the condition

$$(x := E; (P \lhd b(x) \rhd Q)) = ((x := E; P) \lhd b(E) \rhd (x := E; Q)).$$

(6) A conditional joining two assignments (to the same variables) may be replaced by a single assignment of a conditional expression to the same variables:

$$((x := E) \lhd b \rhd (x := F)) = (x := (E \lhd b \rhd F)).$$

(7) The conditional distributes down to the individual components of a list of expressions:

$$(e, E) \lhd b \rhd (f, F) = (e \lhd b \rhd f), (E \lhd b \rhd F).$$

(8) Using these laws, we can eliminate conditionals from sequences of assignments by driving them into the expressions. For example,

$$(x := E; (x := F(x) \lhd b(x) \rhd x := G(x)))$$
$$= (x := (F(E) \lhd b(E) \rhd G(E))).$$

The following theorem will also be useful in reduction to normal forms:

(9) $((x := E \lhd b \rhd \perp); (x := F(x) \lhd c(x) \rhd \perp))$
$= (x := F(E) \lhd c(E) \lhd b \rhd \text{false} \rhd \perp)$.

PROOF.

LHS = $(x := E; ((x := F(x) \lhd c(x) \rhd \perp) \lhd b \rhd (\perp; (x := F(x) \lhd c(x) \rhd \perp)))$

(by (5) under "Sequential Composition")

$= (x := F(E) \lhd c(E) \rhd \perp) \lhd b \rhd \perp$

(by (1) and (5), and (3) under "Sequential Composition")

= RHS

(by (2) and (6) under "The Conditional"). □

Undefined Expressions

If the notations of the programming language include expressions that may be undefined for some values of their operands, then some of the laws quoted above need to be slightly weakened. We assume that the range of expressions allowed in the programming language is sufficiently narrow that for each expression (or list of expressions) E there is a Boolean expression (which we will denote $\mathscr{D}$E) that is true in all circumstances in which evaluation of E would succeed and false in all circumstances in which evaluation of E would fail. Thus evaluation of $\mathscr{D}$E itself will always succeed (this would not be possible in a language with arbitrary programmer-defined functions). Note that $\mathscr{D}$ is not assumed to be a notation of the programming language. Here are some examples:

$$\mathscr{D}\,\text{true} = \mathscr{D}\,\text{false} = \text{true}$$

$$\mathscr{D}(E + F) = \mathscr{D}E \wedge \mathscr{D}F$$

$$\mathscr{D}(E/F) = \mathscr{D}E \wedge \mathscr{D}F \wedge F \neq 0$$

$$\mathscr{D}(E \lhd b \rhd F) = \mathscr{D}b \wedge (\mathscr{D}E \lhd b \rhd \mathscr{D}F)$$

$$\mathscr{D}\mathscr{D}E = \text{true}.$$

Now we stipulate that the effect of attempting to evaluate an expression outside its domain is wholly arbitrary, so

(1) $x := E = (x := E \lhd \mathscr{D}E \rhd \bot)$,

(2) $P \lhd b \rhd Q = (P \lhd b \rhd Q) \lhd \mathscr{D}b \rhd \bot$, and

(3) $P \lhd b \rhd \bot = P \lhd b \lhd \mathscr{D}b \rhd \text{false} \rhd \bot$.

In view of this, some of the preceding laws need alteration, as follows:

(4) $P \lhd b \rhd P = P \lhd \mathscr{D}b \rhd \bot$

(see (3) under "The Conditional").

(5) $(P \lhd b \rhd Q) \lhd c \rhd R$

$= ((P \lhd c \rhd R) \lhd b \rhd (Q \lhd c \rhd R)) \lhd \mathscr{D}b \rhd (\bot \lhd c \rhd R)$

(see (11) under "The Conditional").

(6) $(x := E; x := F(x)) = (x := F(E) \lhd \mathscr{D}E \rhd \bot)$

(see (1) under "Assignment").

(7) $x := E; (x := F(x) \lhd b(x) \rhd x := G(x))$

$= x := (F(E) \lhd b(E) \rhd G(E)) \lhd \mathscr{D}E \rhd \bot$

(see (8) under "Assignment").

Theorem (9) under "Assignment" also requires modification and the proof (but not the statement) of (12) under "The Conditional."

Reasoning with undefined expressions can be complicated and needs some care. But there are also some rewards. For example, the fact that the minimum of an empty set is undefined permits a simple formulation of Dijkstra's linear search theorem [4, pp. 105–106]:

(8) $(i := 0; \neg b(i) * (i := i + 1)) = (i := \min\{i | b(i) \wedge i \geq 0\})$.

Note that, in general, the minimum function appearing on the right-hand side cannot be implemented or included in any programming language. So the right-hand side of (8) should be regarded as an exact specification of the program on the left.

Normal Form

To illustrate the power of the laws given so far, we can use them to reduce every finite program text of our language to a simple normal form. A finite program text is one that does not contain iteration or recursion. In normal form a program looks like

$$\left(\bigcup_{i \leq n} x := E_i\right) \lhd b \rhd \bot,$$

where $b \Rightarrow \mathscr{D}E_i$ for all $i \leq n$, and $\Rightarrow$ denotes logical implication. Without loss of generality, we can ensure that in this context $\mathscr{D}b$ = true by replacing b if necessary by

$$\lhd b \lhd \mathscr{D}b \rhd \text{false} \rhd$$

(by (3) under "Undefined Expressions").

A notable feature of the normal form is that the sequential composition operator does not appear in it.

To explain how to reduce a program text to normal form, it is sufficient to show how each primitive command can be written in normal form and how each operator, when applied to operands in normal form, yields a result expressible in normal form. The section on assignment explained how all assignments of a program can be adapted so that they all have the same list of variables on the left; so we can assume this has already been done.

(1) *SKIP.*

$II = ((x := x) \lhd \text{true} \rhd \bot)$

(by (2) under "Assignment" and (1) under "The Conditional").

(2) *ABORT.*

$\bot = (x := x \lhd \text{false} \rhd \bot)$

(by (2) under "The Conditional").

(3) *Assignment.*

$(x := E) = (x := E \lhd \mathscr{D}E \rhd \bot)$

(by (1) under "Undefined Expressions").

(4) *Nondeterminism.*

$(P \lhd b \rhd \bot) \cup (Q \lhd c \rhd \bot)$

$= (P \cup (Q \lhd c \rhd \bot)) \lhd b \rhd (\bot \cup (Q \lhd c \rhd \bot))$

(by (10) under "The Conditional")

$= ((P \cup Q) \lhd c \rhd (P \cup \bot)) \lhd b \rhd \bot$

(by (10) under "The Conditional," and (1) and (4) under "Nondeterminism")

$= ((P \cup Q) \lhd c \rhd \bot) \lhd b \rhd \bot$

(by (1) and (4) under "Nondeterminism")

$= (P \cup Q) \lhd c \lhd b \rhd \text{false} \rhd \bot$

(by (2) and (6) under "The Conditional").

Here, P and Q stand for lists of assignments separated by ∪, so P ∪ Q is just the union of these two lists. The condition $c \lhd b \rhd \text{false}$ is equivalent to $(c \wedge b)$. Since the operands are normal forms, this is defined everywhere and implies that all expressions in P ∪ Q are also defined.

(5) *Conditional.*

$(P \lhd c \rhd \bot) \lhd b \rhd (Q \lhd d \rhd \bot)$

$= (P \lhd b \rhd Q) \lhd c \lhd b \rhd d \rhd \bot$

(by (12) under "The Conditional").

If

$$P = \bigcup_{i \le n} x := E_i \quad \text{and} \quad Q = \bigcup_{j \le m} x := F_j,$$

then

$$P \lhd b \rhd Q = \bigcup_{i \le n} \bigcup_{j \le m} (x := E_i \lhd b \rhd x := F_j)$$

(by (9) under "The Conditional")

$$= \bigcup_{i \le n} \bigcup_{j \le m} x := (E_i \lhd b \rhd F_j)$$

(by (6) under "Assignment").

Since $c \Rightarrow \mathscr{D}E_i$ and $d \Rightarrow \mathscr{D}F_j$, it follows that

$$c \lhd b \rhd d \Rightarrow \mathscr{D}(E_i \lhd b \rhd F_j), \quad \text{for all } i \text{ and } j.$$

Thus the LHS of (5) is reducible to normal form.

(6) *Sequential composition.*

$$\left(\left(\bigcup_{i \le n} x := E_i\right) \lhd b \rhd \bot\right); \left(\left(\bigcup_{j \le m} x := F_j(x)\right) \lhd c(x) \rhd \bot\right)$$

can be reduced (by distribution through ∪) to

$$\bigcup_{i \le n} \bigcup_{j \le m} ((x := E_i \lhd b \rhd \bot); (x := F_j(x) \lhd c(x) \rhd \bot))$$

$$= \bigcup_{i \le n} \bigcup_{j \le m} (x := F_j(E_i) \lhd c(E_i) \lhd b \rhd \text{false} \rhd \bot)$$

(by (9) under "Assignment").

The method described in (4) can be used to distribute the unions into the conditional, obtaining

$$\left(\bigcup_{i \le n} \bigcup_{j \le m} x := F_j(E_i)\right) \lhd \left(\bigwedge_{i \le n} c(E_i)\right) \lhd b \rhd \text{false} \rhd \bot,$$

where the conjunction notation ∧ can be defined by induction:

$$\bigwedge_{i \le 0} c_i = c_0$$

$$\bigwedge_{i \le n+1} c_i = \left(\bigwedge_{i \le n} c_i\right) \lhd c_{n+1} \rhd \text{false}.$$

This completes the proof that all finite programs are reducible.

The importance of normal forms is that they provide a complete test as to whether two finite program texts denote the same program. The two programs are first reduced to normal form. If the normal forms are equal, so are the programs; otherwise they are unequal.

Two normal forms

$$\left(\bigcup_{i \le n} x := E_i\right) \lhd b \rhd \bot \quad \text{and} \quad \left(\bigcup_{j \le m} x := F_j\right) \lhd c \rhd \bot$$

are equal if and only if

$$b = c$$

and

$$\{v \mid \exists i \le n.\ v = E_i\} = \{w \mid \exists j \le m.\ w = F_j\},$$

where these equations must hold for all values of the variables contained in the expressions b, c, E_i, and F_j.

The truth of these equations may not be decidable, as in integer arithmetic, for example. The results shown here establish only relative completeness.

DOMAIN PROPERTIES

In this section we introduce iteration and recursion, using the methods of [12].

The Ordering Relation

As a preliminary we shall explore the properties of an ordering relation ⊒ between programs.

Definition. $P \sqsupseteq Q \triangleq P \cup Q = P$.

This means that Q is a more deterministic program than P. Everything that Q can do, P may also do, and everything that Q cannot do, P may also fail to do. So Q is, in all respects, a more predictable program and more controllable than P. In any circumstance where P reliably serves some useful purpose, P may be replaced by Q, in the certainty that it will serve the same purpose. But not vice versa. There may be some purposes for which Q is adequate, but P, given its greater nondeterminism, cannot be relied on. Thus $P \sqsupseteq Q$ means that, for any purpose, Q is better than P, or at least as good. From this point on, we will use the comparative "better" by itself, with the understanding that it means "better or at least as good."

The relation ⊒ is not a total ordering on programs, because it is not true for all P and Q that $P \sqsupseteq Q$ or $Q \sqsupseteq P$; P may be better than Q for some purposes, and Q may be better than P for others. However, ⊒ is a *partial* order, in that it satisfies the following laws:

(1) $P \sqsupseteq P$ (reflexivity).

(2) $P \sqsupseteq Q \wedge Q \sqsupseteq P \Rightarrow P = Q$ (antisymmetry).

(3) $P \sqsupseteq Q \wedge Q \sqsupseteq R \Rightarrow P \sqsupseteq R$ (transitivity).

These laws can be proved directly from the definition, together with the laws for ∪.

PROOF.

(1) $P \cup P = P$ (idempotence of $\cup$).

(2) $(P \cup Q = P) \wedge (Q \cup P = Q)$
$\Rightarrow P = P \cup Q = Q \cup P = Q$ (symmetry of $\cup$).

(3) $(P \cup Q = P) \wedge (Q \cup R = Q)$ (antecedent)
$\Rightarrow P \cup R = (P \cup Q) \cup R$ (first antecedent)
$= P \cup (Q \cup R)$ (associativity of $\cup$)
$= P \cup Q$ (second antecedent)
$= P$ (first antecedent). □

The *ABORT* command $\perp$ is the most nondeterministic of all programs. It is the least predictable, the least controllable, and in short, for all purposes, the worst:

(4) $\perp \supseteq P$.

PROOF.

$$\perp \cup P = \perp.$$ □

The machine that behaves either like P or like Q is, in general, worse than both of them:

(5) $(P \cup Q) \supseteq P \wedge (P \cup Q) \supseteq Q$.

PROOF.

$(P \cup Q) \cup P = P \cup (Q \cup P)$ (associativity)
$= P \cup (P \cup Q)$ (symmetry)
$= (P \cup P) \cup Q$ (associativity)
$= P \cup Q$ (idempotence). □

In fact, $P \cup Q$ is the best program that has property (5). Any program R that is worse than both P and Q is also worse than $P \cup Q$, and vice versa:

(6) $R \supseteq (P \cup Q) \equiv (R \supseteq P \wedge R \supseteq Q)$.

PROOF.

LHS $\Rightarrow R \supseteq P$
(by transitivity from (5))
LHS $\Rightarrow R \supseteq Q$
(similarly)
RHS $\Rightarrow (R \cup P = R) \wedge (R \cup Q = R)$
(by definition of $\supseteq$)
$\Rightarrow (R \cup P) \cup (R \cup Q) = R \cup R$
(by adding the equations)
$\Rightarrow R \cup (P \cup Q) = R$
(by properties of $\cup$)
$\Rightarrow$ LHS
(by definition of $\supseteq$). □

If $P \supseteq Q$ this means that Q is, in all circumstances, better than P. It follows that wherever P appears within a larger program, it can be replaced by Q, and the only consequence will be to improve the larger program (or at least to leave it unchanged). For example,

(7) If $P \supseteq Q$ then

$$P \cup R \supseteq Q \cup R$$
$$\wedge (P; R) \supseteq (Q; R)$$
$$\wedge (R; P) \supseteq (R; Q)$$
$$\wedge (P \lhd b \rhd R) \supseteq (Q \lhd b \rhd R)$$
$$\wedge (R \lhd b \rhd P) \supseteq (R \lhd b \rhd Q)$$
$$\wedge (b * P) \supseteq (b * Q).$$

In summary, the law quoted above states that all the operators of our small programming language are *monotonic*, in the sense that they preserve the $\supseteq$ ordering of their operands. In fact, every operation that distributes through $\cup$ is also monotonic.

THEOREM. *If F is any function from programs to programs, and for all programs P and Q, $F(P \cup Q) = F(P) \cup F(Q)$, then F is monotonic.*

PROOF.

$P \supseteq Q \Rightarrow P \cup Q = P$ (by definition of $\supseteq$)
$\Rightarrow F(P) \cup F(Q) = F(P \cup Q)$ (by distribution of F)
$= F(P)$ (by property of $=$)
$\Rightarrow F(P) \supseteq F(Q)$ (by definition of $\supseteq$). □

One important fact about monotonicity is that every function defined from composition of monotonic functions is also monotonic. Since all the operators of our programming language are monotonic, every program composed by means of these operators is monotonic in each of its components. Thus, if any component is replaced by a possibly better one, the effect can only be to improve the program as a whole. If the new program is also more efficient than the old, the benefits are increased.

Least Upper Bounds

We have seen that $P \cup Q$ is the best program *worse* than both P and Q. Suppose we want a program *better* than both P and Q. In general, there will be no such program. Consider the two assignments $x := 1$ and $x := 2$. These programs are incompatible, and there is no program better for all purposes than both. If the final value of x should be 1, the first program is suitable, but the second cannot possibly give an acceptable result. On the other hand, if the final value should be 2, the first program is totally unsuitable.

Consider two nondeterministic programs:

$$P = (x := 1 \cup x := 2 \cup x := 3),$$
$$Q = (x := 2 \cup x := 3 \cup x := 4).$$

In this case there exists a program better than both, namely, $x := 2$. In fact there exists a worst program that

is better than both, and we will denote this by $P \cap Q$:

$$P \cap Q = (x := 2 \cup x := 3).$$

Two programs P and Q are said to be *compatible* if they have a common improvement; and then their worst common improvement is denoted $P \cap Q$. This fact is summarized in the law

(1) $(P \sqsupseteq R) \wedge (Q \sqsupseteq R) \equiv (P \cap Q) \sqsupseteq R.$

Corollary.

$P \sqsupseteq (P \cap Q) \wedge Q \sqsupseteq (P \cap Q)$

(by (1) under "The Ordering Relation").

The operator $\cap$, wherever it is defined, is idempotent, symmetric, and associative, and has identity $\perp$.

(2) $P \cap P = P.$

(3) $P \cap Q = Q \cap P.$

(4) $P \cap (Q \cap R) = (P \cap Q) \cap R.$

(5) $\perp \cap P = P.$

The $\cap$ operator generalizes to any finite set of compatible programs

$$S = \{P, Q, \ldots, T\}.$$

Provided that there exists a program better than all of them, the least such program is denoted $\cap S$:

$\cap S = P \cap Q \cap \cdots \cap T,$

provided $\exists R.P \sqsupseteq R \wedge Q \sqsupseteq R \wedge \cdots \wedge T \sqsupseteq R.$

It follows that

(6) $(\forall P \in S.\ P \sqsupseteq R) \equiv \cap S \sqsupseteq R,$

provided $\cap S$ is defined.

It is important to recognize that $\cap$ is not a combinator of our programming language, and that $P \cap Q$ is not strictly a program, even if P and Q are compatible programs. For example, let P be a program that assigns an arbitrary ascending sequence of numbers to an array, and let Q be a program that subjects the array to an arbitrary permutation. Then $P \cap Q$ would be a program that satisfies both these specifications and consequently would sort the array into ascending order. Unfortunately, programming is not that easy. As in other branches of engineering, it is not generally possible to make many different designs, each satisfying one requirement of a specification, and then merge them into a single product satisfying all requirements. On the contrary, the engineer has to satisfy all requirements in a single design. This is the main reason why designs and programs get complicated.

These problems do not arise with pure specifications, which may be freely connected by the conjunction *and*. $P \cap Q$ may be regarded as an *abstract* program, or a specification of a program that accomplishes whatever P accomplishes *and* whatever Q accomplishes, and fails only when *both* P and Q fail. $P \cap Q$ specifies a program that (if it exists) is, for all purposes, better than both P and Q.

In fact, conjunction is the most useful operator for structuring large specifications. It should be included in any language designed for that purpose. Clarity of specification, achieved by using the conjunction operator, is much more important than using a specification language that can be implemented. It is unfortunate that conjunction is so difficult to implement. As a consequence, the formulation of appropriate specifications and the design of programs to meet them (i.e., software engineering) will always be a serious intellectual challenge.

Limits

Suppose S is a nonempty (possibly infinite) set, and for every pair of its members, S actually contains a member better than both. Such a set is said to be *directed.*

Definition. S is directed means that

$$(S \neq \{\}) \wedge \forall P, Q \in S.\ \exists R \in S.\ P \sqsupseteq R \wedge Q \sqsupseteq R.$$

Examples of directed sets are

$\{P\}$	a set with only one member,
$\{P, P \cup Q\}$	since $P \cup Q \sqsupseteq P$ and $P \sqsupseteq P$, and
$\{P, Q, R\}$	where $P \sqsupseteq R \wedge Q \sqsupseteq R$.

If S is finite and directed, then clearly it contains a member that is better than all the other members, so $\cap S$ is defined and $\cap S \in S$.

If S is directed but infinite, then it does not necessarily contain a best member. Nevertheless, the set has a limit $\cap S$. As noted it is the worst program better than all members of S. The set S is like a convergent sequence of numbers, tending toward a limit that is not a member of the sequence. By selecting members of the set, it is possible to approximate its limit as closely as we please. Since no metric has been introduced, "closeness" has to be interpreted indirectly; that is, for any *finite* program P worse than $\cap S$ there exists a member of S better than P.

One interesting property of the limit of a directed set of programs is that it is preserved by all the operators of our programming language; such operators are therefore said to be *continuous.*

(1) $(\cap S) \cup Q = \cap\{P \cup Q \mid P \in S\}.$

(2) $(\cap S) \lhd b \rhd Q = \cap\{P \lhd b \rhd Q \mid P \in S\}.$

(3) $(\cap S); Q = \cap\{P; Q \mid P \in S\}.$

(4) $Q; (\cap S) = \cap\{Q; P \mid P \in S\}.$

(5) $b * (\cap S) = \cap\{b * P \mid P \in S\}.$

A fact about continuity is that any composition of continuous functions is also continuous. Let X stand for a program, and let $F(X)$ be a program constructed solely by means of continuous operators, and possibly containing occurrences of X. If S is directed, it follows that

(6) $F(\cap S) = \cap\{F(X) \mid X \in S\}$.

Iteration and Recursion

Given a program P and a Boolean expression b, we can define by induction an infinite set

$$\{Q_n \mid n \geq 0\},$$

where

$$Q_0 = \bot,$$

$$Q_{n+1} = (P; Q_n) \lhd b \rhd \text{II}, \quad \text{for all } n \geq 0.$$

From these definitions it is clear that Q_n is a program that behaves like (b * P) up to n iterations of the body P, but breaks on the nth iteration, and can do anything ($\bot$). Clearly, therefore

$$Q_n \supseteq Q_{n+1}, \quad \text{for all } n,$$

(which can be proved formally by induction). Consequently, the set $\{Q_n \mid n \geq 0\}$ is directed, and by taking n large enough, we can approximate as closely as we please to the behavior of the loop (b * P). The loop itself can be defined as the limit of all its approximations:

(1) $b * P = \cap\{Q_n \mid n \geq 0\}$.

The same technique can be used to define a more general form of recursion. Let X stand for the name of the recursive program that we wish to construct, and let $F(X)$ define the intended behavior of the program. Within $F(X)$, each occurrence of X stands for a call on the whole recursive program again. As before, we can construct a series of approximations to the behavior of the recursive program:

$$F^0(Q) = Q,$$

$$F^{n+1}(Q) = F(F^n(Q)), \quad \text{for all } n \geq 0.$$

$F^n(\bot)$ behaves correctly provided that the recursion depth does not equal or exceed n. Because F is monotonic and $F^0(\bot) \supseteq F^1(\bot)$, it follows that

$$F^n(\bot) \supseteq F^{n+1}(\bot), \quad \text{for all } n.$$

Consequently, $\{F^n(\bot) \mid n \geq 0\}$ is a directed set, and we define the recursive program (denoted by $\mu X.F(X)$) as its limit:

(2) $\mu X.F(X) = \cap\{F^n(\bot) \mid n \geq 0\}$.

In accordance with the explanation given above, iteration is a special case of recursion:

(3) $b * P = \mu X.(P; X) \lhd b \rhd \text{II}$.

The most important fact about a recursively defined program is that each of the recursive calls is equal to the whole program again, or more formally that $\mu X.F(X)$ is a solution of the equation $X = F(X)$. This is stated in the following law:

(4) $\mu X.F(X) = F(\mu X.F(X))$.

PROOF.

$$\begin{aligned}
\text{RHS} &= F(\cap\{F^n(\bot) \mid n \geq 0\}) \\
&\quad \text{(by definition of } \mu) \\
&= \cap\{F(F^n(\bot)) \mid n \geq 0\} \\
&\quad \text{(by continuity of } F) \\
&= \cap(\{F^{n+1}(\bot) \mid n \geq 0\} \cup \{\bot\}) \\
&\quad \text{(by definition of } F^{n+1} \text{ and (5)} \\
&\qquad\qquad \text{under "Least Upper Bounds")} \\
&= \cap\{F^n(\bot) \mid n \geq 0\} \\
&\quad \text{(since } F^0(\bot) = \bot) \\
&= \text{LHS} \\
&\quad \text{(by definition).}
\end{aligned}$$

□

Corollary. $b * P = (P; (b * P)) \lhd b \rhd \text{II}$.

In general, there will be more than one solution of the equation $X = F(X)$. Indeed, for the equation $X = X$, absolutely every program is a solution. But, of all the solutions, $\mu X.F(X)$ is the worst:

(5) $Y = F(Y) \Rightarrow \mu X.F(X) \supseteq Y$.

PROOF.

$$\begin{aligned}
(Y = F(Y)) &\Rightarrow (\bot \supseteq Y) \wedge (Y = F(Y)) \\
&\Rightarrow (F(\bot) \supseteq F(Y)) \wedge (Y = F(Y)), \\
&\qquad\qquad \text{(by } F \text{ monotonic)} \\
&\Rightarrow (F(\bot) \supseteq Y).
\end{aligned}$$

□

By induction it follows that, for any $n \geq 0$,

$$\begin{aligned}
Y = F(Y) &\Rightarrow F^n(\bot) \supseteq F^n(Y) \wedge Y = F^n(Y) \\
&\Rightarrow F^n(\bot) \supseteq Y \\
&\Rightarrow \cap\{F^n(\bot) \mid n \geq 0\} \supseteq Y \\
&\qquad \text{(by (13) under "Least Upper Bounds")}
\end{aligned}$$

as required.

SPECIFICATIONS

Two important concepts have been introduced thus far. The first is that a specification describes the intended behavior of a program, without giving any guidance as to how the program might be executed. Secondly, a concrete program P may be *better* than a specification S; so whenever a program that behaves like S is required, the concrete program P will serve the purpose. In this case, we can say that P *satisfies* the specification S, or in symbols, $S \supseteq P$. It is the responsibility of the programmer, when given a specification S, to find a program P that satisfies S. The practical purpose of the laws presented is to assist in this task.

Introduced here is a calculus of specifications to aid in the development of programs. Specifications do not have to be executed by machine, so there is no reason

to confine ourselves to the notations of a particular programming language. Also, it is not necessary to confine ourselves to specifications that can be satisfied. As an extreme example, we introduce the specification $\top$, which cannot be satisfied by any program whatsoever.

To accept the risk of asking the impossible has as its reward that the $\cap$ operator is defined on all specifications: Wherever R and S are inconsistent, the result of $(R \cap S)$ is $\top$. Furthermore, if S is *any* set of specifications, then

$\cap S$ is the specification that requires *all* R in S to be satisfied, and
$\cup S$ is the specification that requires *some* R in S to be satisfied.

These specifications are limits of S:

(1) $Q \supseteq \cup S \equiv \forall R \in S.\ Q \supseteq R.$

(2) $\cap S \supseteq Q \equiv \forall R \in S\ \ R \supseteq Q.$

The $\supseteq$ ordering applies to specifications, just as it does to programs, but it can be interpreted in a new sense. If $S \supseteq R$, it means that S is a weaker specification and easier to meet than R. Any program that satisfies R will serve for S, but it may be that more programs will satisfy S. Thus $\bot$ is the easiest specification, satisfied by any program, and $\top$ is impossible.

We do not introduce any specific notation or language for specifications; we permit any language that describes a relationship between the values of variables before and after execution of a program. Thus we may use the notations of a programming language, possibly extended even by noncomputable operators, or we may use predicates as in predicative programming [6] or predicate pairs as in VDM [9]. We assume that the specification language includes at least all the notations of our programming language, so that a program is its own strongest specification.

Thus all programs are specifications, but not necessarily vice versa. As a consequence, there are certain laws that are true for programs, but *not* for specifications. In particular, law (3) in "Sequential Composition" and law (4) in "Limits" are *not* valid for specifications. However, we believe it is reasonable to insist that specifications obey all the laws of the calculus of relations [13].

Weakest Prespecification

Specifications may be constructed in terms of all the operators available for concrete programs. For example, if R and S are specifications, then (R; S) is a specification satisfied by any program (P; Q), where P satisfies R and Q satisfies S (it can also be satisfied by other programs). This fact is extremely useful in the top-down development of programs (also known as stepwise refinement). Suppose, for example, that the original task is to construct a program that meets the specification W. Perhaps we can think of a way to decompose this task into two simpler subtasks specified by R and S. The correctness of the decomposition can be proved by showing that $W \supseteq R; S$. This proof should be completed before embarking on design for the subtasks R and S. Then similar methods can be used to find programs P and Q that solve these subtasks, such that $R \supseteq P$ and $S \supseteq Q$. It follows immediately from monotonicity of sequential composition that P; Q is a program that will solve the original task W, that is, $W \supseteq (P; Q)$.

In approaching the task W, suppose we think of a reasonable specification for the second of the two subtasks S, but we do not know the exact specification of the first subtask. It would be useful to calculate R from S and W. Therefore we define the *weakest prespecification* $S\backslash W$ to be the weakest specification that must be met by the first subprogram R in order that the composition (R; S) will accomplish the original task W. This fact is expressed in symbols:

(1) $W \supseteq (S\backslash W); S.$

$(S\backslash W)$ is a sort of left quotient of W by S; the divisor S can be canceled by postmultiplication, and the result will be the same as W or better.

Here are some examples of weakest prespecifications, where x is an integer variable:

$(x := 3 \times x)\backslash(x := 6 \times y) = (x := 2 \times y),$

because $(x := 2 \times y;\ x := 3 \times x) = (x := 6 \times y).$

$(x := 2 \times x)\backslash(x := 3) = \top,$

since 3 is odd and cannot be the result of doubling an integer.

$(x := 2 \times x)\backslash(x := 3 \cup x := 4) = (x := 2),$

because $(x := 3 \cup x := 4) \supseteq x := 4 = (x := 2;\ x := 2 \times x).$

The law given above does not uniquely define $S\backslash W$. But, of all the solutions for X in the inequality $W \supseteq (X; S)$, the solution $S\backslash W$ is the easiest to achieve. Thus, to find such a solution, a necessary and sufficient condition is that the solution should satisfy $S\backslash W$:

(2) $W \supseteq (X; S) \equiv (S\backslash W) \supseteq X.$

Thus in developing a sequential program to meet specification W, there is no loss of generality in taking $S\backslash W$ as the specification of the left operand of sequential composition, given that S is the specification of the right operand. That is why it is called the *weakest* prespecification. For the remainder of this article, for convenience we will omit the word *weakest*.

The prespecification $P\backslash R$, where P is a program, plays a role very similar to Dijkstra's weakest precondition. It satisfies the analogue of several of his conditions. In the following three laws, P must be a *program*.

(3) To accomplish an impossible task, it is still impossible, even with the help of P:

$$P\backslash\top = \top.$$

(4) To accomplish two tasks with the help of P, one must write a program that accomplishes both of them simultaneously:

$$P\backslash(R1 \cap R2) = (P\backslash R1) \cap (P\backslash R2).$$

This distributive law extends to limits of arbitrary sets:

$$P\backslash(\cap S) = \cap\{P\backslash R \mid R \in S\}.$$

(5) Finally, consider a set of specifications $S = \{R_i \mid i \geq 0\}$ such that

$$R_{i+1} \supseteq R_i.$$

Then

$$P\backslash(\cup S) = \cup\{P\backslash R_i \mid i \geq 0\}.$$

The following laws are very similar to the corresponding laws for weakest preconditions.

(6) The program II changes nothing. Anything one wants to achieve after II must be achieved before:

$$II\backslash R = R.$$

(7) To achieve R with the aid of $P \cup Q$, one must achieve it with either of them:

$$(P \cup Q)\backslash R = (P\backslash R) \cap (Q\backslash R).$$

(8) To achieve R with the aid of (P; Q), one must achieve $(Q\backslash R)$ with the aid of P:

$$(P; Q)\backslash R = P\backslash(Q\backslash R).$$

(9) The corresponding law for the conditional requires a new operator on specifications:

$$(P \lhd b \rhd Q)\backslash R = (P\backslash R) \lhd b \rhd (Q\backslash R),$$

where $S \lhd b \rhd T$ specifies a program as follows: If b is true *after* execution, it has behaved in accordance with specification S. If b is false afterwards, it has behaved in accordance with specification T. $P \lhd b \rhd Q$ is not a program, even if P and Q are; in fact it may not even be implementable. Consider the example

$$x := \text{false} \lhd x \rhd x := \text{true}.$$

General Inverse

The $\backslash$ operator has a dual, /. (R/S) is the weakest specification of a program X such that $R \supseteq (S; X)$. Its properties are very similar to those of $\backslash$; for example,

(1) $R \supseteq S; (R/S)$,

(2) $R \supseteq (S; X) \equiv (R/S) \supseteq X$,

(3) $T/P = T$, if P is a program,

(4) $(R1 \cap R2)/P = (R1/P) \cap (R2/P)$,

(5) $R/II = R$,

(6) $R/(P \cup Q) = (R/P) \cap (R/Q)$, and

(7) $R/(P; Q) = (R/P)/Q$.

The prespecification and postspecification are, in a sense, the right and left inverses of sequential composition. This type of inverse can be given for any operator F that distributes through arbitrary unions. It is defined as follows:

(8) $F^{-1}(R) = \cup\{P \mid R \supseteq F(P)\}$.

This is not an exact inverse of F, but it satisfies the law

(9) $R \supseteq F(F^{-1}(R))$.

PROOF.

$$\begin{aligned} RHS &= F(\cup\{P \mid R \supseteq F(P)\}) && \text{(by definition of } F^{-1}) \\ &= \cup\{F(P) \mid R \supseteq F(P)\} && \text{(by distribution of } F) \\ &\subseteq R && \text{(by set theory).} \end{aligned}$$ □

Since $F^{-1}(R)$ is the union of all solutions for X in the inequation $R \supseteq F(X)$, it must be the weakest (most general) solution:

(10) $R \supseteq F(X) \equiv F^{-1}(R) \supseteq X$.

The condition that F must distribute through $\cup$ is essential to the existence of the inverse F^{-1}. To show this, consider the counterexample:

$$F(X) = X; X$$
$$P = x := x$$
$$Q = x := -x.$$

F is a function that may require more than one execution of its operand. When applied to the nondeterministic choice of two programs P or Q, each execution may produce a different choice. Consequently, F does not distribute, as shown by the following example:

$$\begin{aligned} F(P \cup Q) &= (P \cup Q); (P \cup Q) \\ &\qquad \text{(by definition of } F) \\ &= (P; P) \cup (P; Q) \cup (Q; P) \cup (Q; Q) \\ &\qquad \text{(by ; disjunctive)} \\ &= (x := x; x := x) \cup (x := x; x := -x) \\ &\quad \cup (x := -x; x := x) \cup (x := -x; x := -x) \\ &= x := x \cup x := -x. \end{aligned}$$

But

$$\begin{aligned} F(P) \cup F(Q) &= (x := x; x := x) \cup (x := -x; x := -x) \\ &= x := x. \end{aligned}$$

Since $P \supseteq F(P)$ and $P \supseteq F(Q)$, it follows that

$$\cup\{X \mid P \supseteq F(X)\} \supseteq P \cup Q \qquad \text{(by set theory).}$$

By law (10) and the definition of $F^{-1}(P)$, we could conclude that $P \supseteq F(P \cup Q)$, which is false. The contradiction shows that F does not have an inverse, even in the weak sense described by law (10).

The inverse $F^{-1}(R)$ (when it exists) could be of assistance in the top-down development of a program to meet the specification R. Suppose it is decided that the top-level structure of the program is defined by F. Then it will be necessary to calculate $F^{-1}(R)$ and use it as the specification of the component program X, secure in the knowledge that the final program $F(X)$ will meet the original specification R: That is, $R \supseteq F(X)$.

Unfortunately, the method does not generalize to a structure F with two or more components. Therefore it would be necessary to fix all but one of the components before calculating the inverse.

CONCLUSION

The laws presented here should assist programmers in the reasoning necessary to develop programs that meet their specifications. They should also help with optimization by algebraic transformation. The basic insight is that programs themselves, as well as their specifications, are mathematical expressions. Therefore they can be used directly in mathematical reasoning in the same way as expressions that denote familiar mathematical concepts, such as numbers, sets, functions, groups, categories, among others. It is also very convenient that programs and specifications are treated together in a homogeneous framework; the main distinction between them is that programs are a subclass of specification expressed in such severely restricted notations that they can be input, translated, and executed by a general-purpose digital computer.

However, we admit the exposition of this article does have deficiencies. One theoretical weakness is that the laws are presented as self-evident axioms or postulates, intended to command assent from those who already understand the properties of programs they express.

For sequential programs and their specifications, the relevant mathematical definitions would be formulated within the classical theory of relations [7]. The use of such definitions to prove the laws enumerated in this article yields a valuable reassurance that the laws are consistent. Furthermore, the definitions give additional insight into the mathematics of programming and how it may be applied in practice. Specifically they suggest additional useful laws, and establish that a given set of laws is *complete* in the sense that some clearly defined subset of all truths about programming can be deduced directly from the laws, without appeal to the possibly greater complexity of the definitions. This could be extremely useful to the practicing programmer, who does not have to know the foundations of the subject any more than the scientist has to know about the definition of real numbers in terms of Dedekind cuts.

Although nearly one hundred laws are given in this article, we are still a long way from knowing how to apply them directly to the design of correct and efficient programs on the scale required by modern technology. Gaining practical experience in the application of these mathematical laws to programming is the way to go. The search for deeper and more specific theorems that can be used more simply on limited but not too narrow ranges of problems should continue. That is the way that applied mathematics, as well as pure mathematics, has made such great progress in the last two thousand years. If we follow this example, perhaps we may make more rapid progress, both in theoretical research and in its practical application.

REFERENCES

Note: References [1], [3], and [5] are not cited in text.

1. Backhouse, R.C. *Program Construction and Verification.* Prentice-Hall International, London, 1968.
2. Backus, J. Can programming be liberated from the von Neumann style? *Commun. ACM 21*, 8 (Aug. 1978), 613–641.
3. de Bakker, J.W. *Mathematical Theory of Program Correctness.* Prentice-Hall International, London, 1980.
4. Dijkstra, E.W. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, N.J., 1976.
5. Gries, D. *The Science of Programming.* Springer-Verlag, New York, 1981.
6. Hehner, E.C.R. Predicative programming parts I and II. *Commun. ACM 27*, 2 (Feb. 1984), 134–151.
7. Hoare, C.A.R., and He, J. Weakest prespecification. Tech. Monogr. PRG-44, Programming Research Group, Oxford Univ., 1985.
8. Igarishi, S. An axiomatic approach to equivalence problems of algorithms with applications. Rep., Computer Centre, Univ. of Tokyo, 1968.
9. Jones, C.B. *Software Development: A Rigorous Approach.* Prentice-Hall International, London, 1980.
10. Kowalski, R.A. The relation between logic programming and logic specification. In *Mathematical Logic and Programming Languages*, C.A.R. Hoare and J.C. Shepherdson, Eds. Prentice-Hall International, London, 1985, pp. 11–27.
11. Roscoe, A.W. Laws of Occam programming. Tech. Monogr. PRG-53, Programming Research Group, Oxford Univ., 1986.
12. Scott, D.S. Outline of a mathematical theory of computation. Tech. Monogr. PRG-2, Programming Research Group, Oxford Univ., 1970.
13. Tarski, A. On the calculus of relations. *J. Symbolic Logic 6* (1941), 73–89.

CR Categories and Subject Descriptors: D.1.4 [**Programming Techniques**]: Sequential Programming; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*semantics*; D.3.4 [**Programming Languages**]: Processors—*optimization*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*pre- and post-conditions*; *specification techniques*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*algebraic approaches to semantics*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program transformation*

General Terms: Design, Languages, Theory

Authors' Present Addresses: C.A.R. Hoare, Dept. of Computer Science, Taylor Hall 2.2124, University of Texas at Austin, Austin, TX 78712; He Jifeng, C.C. Morgan, A.W. Roscoe, J.W. Sanders, I.H. Sorensen, J.M. Spivey, and B.A. Sufrin, Programming Research Group, Oxford University Computing Laboratory, 11, Keble Road, Oxford OX1 3GQ, England; I.J. Hayes, Dept. of Computer Science, University of Queensland, St. Lucia, Queensland, Australia, 4067.

A few last minute corrections arrived too late for inclusion here. They will be printed in CACM, September 1987.