LAWS OF PROGRAMMING

A TUTORIAL PAPER

by

C.A.R.Hoare, He Jifeng, I.J.Hayes,
C.C.Morgan, J.W.Sanders, I.H.Sørensen,
J.M.Spivey, B.A.Sufrin, A.W.Roscoe

Technical Monograph PRG-45

May 1985

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

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

LAWS OF PROGRAMMING

A TUTORIAL PAPER

C.A.R. Hoare, He Jifeng, I.J. Hayes, C.C. Morgan,
J. Sanders, I.H. Sorensen, J.M. Spivey, B.A. Sufrin, A.W. Roscoe

Summary

A complete set of algebraic laws is given for Dijkstra's non-deterministic 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.

Warning

In many programming languages use of these laws of programming may lead to error. You are advised to consult your language definition and implementation manuals to determine the circumstances in which their use is valid.

CONTENTS

1.	Introduction	1
1.1	The Language	4
1.2	Summary	8
1.3	Examples	10
2.	Algebraic Laws	11
2.1	Nondeterminism	11
2.2	Conditional	13
2.3	Sequential Composition	14
2.4	Assignment	16
2.5	Undefined expressions	18
2.6	Normal form	19
3.	Domain properties	22
3.1	The ordering relation	22
3.2	Least upper bounds	25
3.3	Limits	29
3.4	Iteration and recursion	30
4.	Specifications	33
4.1	Weakest prespecification	34
4.2	General inverse	38
5.	Conclusion	41
	References	43

1. Introduction

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

(1) Multiplication is symmetric, or in symbols

$$x \times y = y \times x \qquad \text{for all numbers } x \text{ and } y$$

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

$$0 \times x = 0$$

(4) Multiplication by 1 leaves a number unchanged

$$1 \times x = x$$

(5) Division is the inverse of multiplication

$$y \times (x/y) = x \qquad \text{provided } y \neq 0$$

If multiplication were not symmetric, we would also need a left quotient operator $\backslash$, satisfying the law

$$(y \backslash x) \times y = x \qquad \text{provided } y \neq 0$$

Another law relating multiplication and division is

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

(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.

If multiplication were not symmetric, we would distinguish distribution to the left (described above) from distribution to the right

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

An operator is distributive through another if it distributes both to the left and to the right.

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

$$x \leq y \implies x \times z \leq y \times z \qquad \text{provided } z \geq 0$$

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

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

(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

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

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

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

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

Any mathematician or anginser will be intimately familiar with all these laws (and many more); and he will use them frequently and almost instinctively, without noticing he has done so. 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 ever a single law. An unsympathetic observer might claim that programmers are such an undisciplined bunch that they would not obey such laws, even if they knew them. Some computer scientists have despaired of finding rational laws to govern conventional procedural programming, and recommend instead the use of functional programming [Backus] or logic programming [Kowalski].

In this paper, we shall substantiate a claim that conventional procedural programs are mathematical expreasions, 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.

1.1 The language

In order to formulate mathematical laws, it is necessary to introduce some notation for describing programs. I shall use a notation (programming language) which is especially concise and suitable for its purpose, based on the language introduced in [Dijkstra]. It has three kinds of primitive commanc, and five methods of composing commands into larger commands (programs).

(1) SKIP

The SKIP command is denoted II; execution of this command terminates successfully, leaving everything unchanged.

(2) ABORT

The ABORT command is danoted $\perp$; it places no constraint on the behaviour or misbehaviour of the executing machine, which may do anything, or fail to do anything; in particular, it may fail to terminate. Thus $\perp$ represents the behaviour of a broken machine, or a program that has run wild. This is certainly a repugnant program, but it plays an important role in the theory and its application. A programmer has a duty not to write a program that runs wild; in order to prove absence of this error, one needs a mathamatical theory that includes its presence.

(3) Assignment

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-effect, and stipulate that the values of the variables in the list x do not change until ell the evaluations are complete. For simplicity, we shall also essume 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 2.5.

(4) Sequential Composition

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

(5) 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

$$\underline{\text{if}}\ \ b\ \ \underline{\text{then}}\ \ P\ \ \underline{\text{else}}\ \ Q$$

We have chosen an infix notation, $\lhd b \rhd$ because it simplifies expression of the relevant algebraic laws.

(6) Non-determinism

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

(7) 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

$$\underline{\text{while}}\ \ b\ \ \underline{\text{do}}\ \ P$$

(8) Recursion

Let X be the name of a recursively defined program, and let F(X) (containing occurrences of the name X) be a program defining its behaviour. Then $\mu X.F(X)$ is the program which behaves like $F(\mu X.F(X))$; i.e. all recursive occurrences of the program name have been replaced by the whole recursive program. Of course, iteration is only a special case of recursion

$$b*P \ = \ \mu X.(P;X) \lhd b \rhd \mathrm{I1}$$

Iteration is simpler and more familiar than general recursion, and so it is worth treating seperately.

As an example of the use of these notations, here is a program which computes the quotient q and remeinder r of division of non-negative x by positive y. It offers a choice of methods, one of which terminates when y = 0

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

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

This example illustrates a suggested order of precedence for program combinators

, binds tightest

:=

*

;

$\not\vdash b \dashv$

$\cup$ binds loosest

Normal arithmetic operators bind tightsst of all. But for the benefit of a readar it is kinder to insert at least some of the brackets, for example

$$(q,r := 0,x \; ; \; (r \geq y * (q,r := q+1, r-y)))$$
$$\cup((q,r := x \div y, x \;\underline{\text{rem}}\; y) \not\vdash y \neq 0 \dashv q := 0)$$

The notations of our language can be defined in terms of E.W. Dijkstra's language of guarded commands

$$P \cup Q = \underline{\text{if}}\ \text{true} \longrightarrow P \;[]\; \text{true} \longrightarrow Q \;\underline{\text{fi}}$$
$$P \not\vdash b \dashv Q = \underline{\text{if}}\ b \longrightarrow P \;[]\; \neg b \longrightarrow Q \;\underline{\text{fi}}$$

where $\neg b$ is the negation of b

$$b * P = \underline{\text{do}}\ b \longrightarrow P \;\underline{\text{od}}$$

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

$$\underline{\text{if}}\ b \longrightarrow P \;[]\; c \longrightarrow Q \;\underline{\text{fi}} = ((P \cup Q) \not\vdash c \dashv P) \not\vdash b \dashv (Q \not\vdash c \dashv \perp)$$
$$\underline{\text{do}}\ b \longrightarrow P \;[]\; c \longrightarrow Q \;\underline{\text{od}} = (b \vee c) * (\underline{\text{if}}\ b \longrightarrow P \;[]\; c \longrightarrow Q \;\underline{\text{fi}})$$

Thus our language is effectively the same as Dijkstra's; the only reason for the slight change of notation is to replace the polyadic notation of "guarded commancs"by binary infix notations, which greatly simplify the formulation of algebraic laws.

1.2 Summary

The laws to be given in this paper apply not only to concrete programs, expressed in the notations of the programming language described in section 1.1; most of them apply also to program specifications, which can be expressed in a considerably 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 the same laws.

(1) Finite programs are expressible in the notations of the programming language, but excluding iteration and recursion. Laws for finite programs are given in section 2. They are sufficiently powerful to permit every finite program to be reduced to a simple normal form. The definition of equality between normal forms extends in this way to all finite programs.

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

(3) Abstract programs are expressad by means of programming notations plus an additional operator for denoting a limit of a convergent set of consistent programs. The relevant concepts and laws are those of domain theory, and they are explained in section 3.

Objects in the first three classes are called programs, and they all satisfy all the laws of programming given in sections 2 and 3.

(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, including even negation. The laws which 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 which cannot be satisfied by any program.

The distinction between these classes may seem complicated; but in fact it is as simple as familiar distinctions made between different classes of number.

(1) Finite programs can be likened to rational numbers. Algebraic laws permit all arithmetic expressions to be reduced to a ratio of co-prime integers, whose equality may 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 non-denumerable set.

(4) Specifications may be likened to complex numbers, for which 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; nevertheless it pays to use them in definition, calculation and proof, even when 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.

1.3 Examples

This paper shows many examples of the practical use of the quoted laws; these examples occur only in the proof of other laws.

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 much deeper mathematics than the elementary algebra presented in this paper. You would not expect to illustrate the laws of arithmetic by a case study in the design of a bridge. Like the laws of arithmetic the laws of programming are broad and shallow: like the grammatical laws of a foreign language, learn them, learn to use them without thinking, and then forget them.

2. Algebraic laws

In this section we shall give about thirty algebraic laws relating to finite programs, i.e., programs that do not contain iterations or recursions, which will be treated in section 3. 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 programs are equal.

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 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.

2.1 Nondeterminism

The laws governing nondeterministic choice apply to all kinds of choice.

(1) Clearly, it does not make any difference in what order such a choice is offered: "milk or cream?" is the same as "cream or milk?"

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

(2) A choice between three alternatives (milk, cream, or brandy) 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 (Hobson's choice) offers no choice at all

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

(4) The abort commano already allows completely arbitrary behaviour, so an offer of further choice makes no difference to it

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

This lew is sometimes known as Sod's law;* the left hand side describes a machine that <u>can</u> go wrong (or can bahave like P); the right hand side might be taken to describe a machine that <u>will</u> go wrong. But the true meaning of the law is actually worse than this: the machine 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 [Murphy].

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

$$\bigcup_{i \leq n} P_i = P_0 \cup P_1 \cup \ldots \cup P_n$$

This is purely a convenient abbreviation, and is not needed in a programming language.

---

* Sod's law states "If it can go wrong it will".

2.2 Conditional

For each given Boolean expression b the choice operator $\lhd b \rhd$ specifies a choice between two alternatives written on each side of it. The first two laws express most clearly the criterion for making this choice, i.e., the truth or falsity of b

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

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

Like $\cup$, 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) $P \lhd b \rhd Q = Q \lhd \overline{b} \rhd P$

where $\overline{b}$ 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 $c \lhd b \rhd d$ is a conditional expression, giving value c if b is true and d if b is false

(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 conditional, or afterwards, 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 $\lhd b \rhd$

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

Proof LHS $= (P \cup Q) \lhd \overline{b} \rhd R = (P \lhd \overline{b} \rhd R) \cup (Q \lhd \overline{b} \rhd R) =$ RHS

An operator that distributes like this through $\cup$ is said to be disjunctive.

Any operation that does not change the value of the Boolean expression b will distribute through $\lhd b \rhd$. 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 $\lhd c \rhd$ distributes through another conditional with a possibly different condition $\lhd b \rhd$.

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

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

$=$ LHS by (7) and (4

2.3 Sequential Composition

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

$P;(Q;R) = (P;Q);R$ 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

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

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

$$(\perp;P) = (P;\perp) = \perp \qquad \text{zero } \perp$$

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

(4) A machine which selects between P and Q and then performs R when the selected alternative terminates cannot be distinguished from one which 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, so ; 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.

2.4 Assignment

It is a law of mathematics that the value of an expression is unchanged when the variables it contains are replaced by their values. If E(x) is a list of expressions, and F is a list of the values of the variables x, than 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

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

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

$$(x:=x) \ = \ II$$

(3) In fact such a vacuous assignment can be added to any other assignment without changing its effect (recall 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

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

(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) $(x := E \lhd b \rhd \bot) ; ((x := F(x)) \lhd c(x) \rhd \bot) = (x := F(E)) \lhd c(E) \lhd b \rhd false \rhd \bot$

Proof LHS $= (x := E ; ((x := F(x) \lhd c(x) \rhd \bot))$
$\lhd b \rhd (\bot ; (x := F(x) \lhd c(x) \rhd \bot)))$ 2.3(5)
$= (x := F(E) \lhd c(E) \rhd \bot) \lhd b \rhd \bot$ (5),(1),2.3(3)
$=$ RHS 2.2(6), 2.2(2)

2.5 Undefined expressions

If the notations of the programming language include expressions which 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 language is sufficiently powerful that it is always possible to test in advance whether evaluation of an expression is going to fail, and that this test itself never fails. Thus for every list of expressions E there is a Boolean expression $\mathcal{D}E$* which gives the answer true in just those circumstances that evaluation of E would be successful. Thus, for example

$$\begin{aligned}
\mathcal{D}\,\mathrm{true} &= \mathcal{D}\,\mathrm{false} = \mathrm{true} \\
\mathcal{D}(E+F) &= \mathcal{D}E \wedge \mathcal{D}F \\
\mathcal{D}(E/F) &= \mathcal{D}E \wedge \mathcal{D}F \wedge F \neq 0 \\
\mathcal{D}(E \lhd b \rhd F) &= \mathcal{D}b \wedge (\mathcal{D}E \lhd b \rhd \mathcal{D}F) \\
\mathcal{D}\mathcal{D}E &= \mathrm{true}
\end{aligned}$$

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

$$\begin{aligned}
&(1)\quad x := E = (x := E \lhd \mathcal{D}E \rhd \bot) \\
&(2)\quad P \lhd b \rhd Q = (P \lhd b \rhd Q) \lhd \mathcal{D}b \rhd \bot \\
&(3)\quad P \lhd b \rhd \bot = P \lhd b \lhd \mathcal{D}b \rhd \mathrm{false} \rhd \bot
\end{aligned}$$

In view of this, the following laws need alteration

$$\begin{aligned}
&(4)\quad P \lhd b \rhd P = P \lhd \mathcal{D}b \rhd \bot && \text{see 2.2(3)} \\
&(5)\quad (P \lhd b \rhd Q) \lhd c \rhd R = ((P \lhd c \rhd R) \lhd b \rhd (Q \lhd c \rhd R)) \lhd \mathcal{D}b \rhd (\bot \lhd c \rhd R) && \text{see 2.2(11} \\
&(6)\quad (x := E\ ;\ x := F(x)) = (x := F(E) \lhd \mathcal{D}E \rhd \bot) && \text{see 2.4(1)} \\
&(7)\quad x := E\ ;\ (x := F(x) \lhd b(x) \rhd x := G(x)) \\
&\qquad = x := (F(E) \lhd b(E) \rhd G(E)) \lhd \mathcal{D}E \rhd \bot && \text{see 2.4 (8)}
\end{aligned}$$

---

* $\mathcal{D}$ is not assumed to be a notation of the programming language.

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 exceptionally simple formulation of Dijkstra's linear search theorem [Dijkstra p. 105-106].

(8) $\quad (i := 0;\ (\overline{b}(i) * i := i+1)) = (i := \min\{i \mid b(i) \wedge i \geq 0\})$

2.6 Normal form

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

$$(\bigcup_{i \leq n} x := E_i)\ \langle b \rangle\ \bot$$

where $b \Longrightarrow \mathcal{D}E_i$ for all $i \leq n$.

Without loss of generality, we can ensure that in this context

$$\mathcal{D}b = \text{true} \quad .$$

by replacing b if necessary by

$$\langle b \rangle \mathcal{D}b \rangle\ \text{false}$$ (see 2.5(3))

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

To show how to reduce a program 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. In section 2.4 we have shown 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

$$II = ((x := x) \lhd true \rhd \bot) \qquad 2.4(2)\ \ 2.2(1)$$

(2) Abort

$$\bot = (x := x \lhd false \rhd \bot) \qquad 2.2(2)$$

(3) Assignment

$$(x := E) = (x := E \lhd \mathcal{D}E \rhd \bot) \qquad 2.5(1)$$

(4) Nondeterminism

$$\begin{aligned}
&(P \lhd b \rhd \bot) \cup (Q \lhd c \rhd \bot) \\
&\quad = (P \cup (Q \lhd c \rhd \bot)) \lhd b \rhd (\bot \cup (Q \lhd c \rhd \bot)) && 2.2(1) \\
&\quad = (P \cup Q) \lhd c \rhd (P \cup \bot)) \lhd b \rhd \bot && 2.2(10)\ \ 2.1(4) \\
&\quad = ((P \cup Q) \lhd c \rhd \bot) \lhd b \rhd \bot && 2.1(4) \\
&\quad = (P \cup Q) \lhd c \lhd b \rhd false \rhd \bot && 2.2(6),\ 2.2(2)
\end{aligned}$$

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

(5) Conditional

$$(P \lhd c \rhd \bot) \lhd b \rhd (Q \lhd d \rhd \bot) = (P \lhd b \rhd Q) \lhd c \lhd b \rhd d \rhd \bot \qquad 2.2(12)$$

If $P = \bigcup_{i \in n} x := E_i$ and $Q = \bigcup_{j \in m} x := F_j$

$$\begin{aligned}
\text{then } P \lhd b \rhd Q &= \bigcup_{i \in n} \bigcup_{j \in m} (x := E_i \lhd b \rhd x := F_j) && 2.2(9) \\
&= \bigcup_{i \in n} \bigcup_{j \in m} x := (E_i \lhd b \rhd F_j) && 2.4(6)
\end{aligned}$$

Since $c \Longrightarrow \mathcal{D}E_i$ and $d \Longrightarrow \mathcal{D}F_j$, it follows that

$$c \{b\} d \Longrightarrow \mathcal{D}(E_i \{b\} F_j) \qquad \text{for all } i \text{ and } j$$

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

(6) Sequential Composition

$$(\bigcup_{i \leq n} x := E_i) \{b\} \perp) ; ((\bigcup_{j \leq m} x := F_j(x)) \{c(x)\} \perp)$$

can be reduced (by distribution through $\cup$) to

$$\bigcup_{i \leq n} \bigcup_{j \leq m} ((x := E_i \{b\} \perp) ; (x := F_j(x) \{c(x)\} \perp))$$

$$= \bigcup_{i \leq n} \bigcup_{j \leq m} (x := F_j(E_i) \{c(E_i) \{b\} \text{false}\} \perp) \qquad \text{by 2.4(9)}$$

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

$$(\bigcup_{i \leq n} \bigcup_{j \leq m} x := F_j(E_i)) \{(\bigwedge_{i \leq n} c(E_i)) \{b\} \text{false}\} \perp$$

where the conjunction notetion $\bigwedge$ can be defined by induction

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

$$\bigwedge_{i \leq n+1} c_i = (\bigwedge_{i \leq n} c_i) \{c_{n+1}\} \text{false}$$

That completes the proof that all finite programs are reducible.

The importance of normal forms is that they provide a complete test whether two finite programs are equal or unequal. 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 $(\bigcup_{i \leq n} x := E_i) \{b\} \perp$ and $(\bigcup_{j \leq m} x := F_j) \{c\} \perp$

are equal if and only if

$$b = c$$

and $$\{v \{b\} \perp \mid \exists i \leq n.\ v = E_i\} = \{w \{c\} \perp \mid \exists j \leq m.\ w = F_j\}$$

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

3. Domain properties

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

3.1 The ordering relation

As a prelimary we shall explore the properties of an ordering relation $\sqsupseteq$ between programs.

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

This means that Q is a more deterministic program than P. Everything that Q can do, P may also do; and everything that Q can fail to 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 for which P, owing to its greater nondeterminism, cannot be relied upon. Thus $P \sqsupseteq Q$ means that for any purpose Q is better than P, or at least as good. In future, we will use the comparative "better" by itself, on the understanding that it means "better or at laast as good".

The relation $\sqsupseteq$ 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, $\sqsupseteq$ is a <u>partial</u> order, in that it satisfies the following laws

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

(2) $P \sqsupseteq Q \wedge Q \sqsupseteq P \Longrightarrow P = Q$ (antisymmetry)

(3) $P \sqsupseteq Q \wedge Q \sqsupseteq R \Longrightarrow P \sqsupseteq R$ (transitivity)

These laws can be proved directly from the definition, together with the laws for $\cup$

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

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

(3) $(P \cup Q = P) \wedge (Q \cup R = Q)$ (antecedent)

$\Longrightarrow P \cup R = (P \cup Q) \cup R$ (first antecedent)

$= P \cup (Q \cup R)$ (associativity $\cup$)

$= P \cup Q$ (second antecedent)

$= P$ (first antecedent)

The abort command $\perp$ is the most nondeterministic of all programs, the least predictable, the least controllable; and in short, for all purposes, it is 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) $(P \cup Q) \supseteq P \quad \wedge \quad (P \cup Q) \supseteq Q$

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 this property. Any program R which 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 $\Longrightarrow R \supseteq P$ by transitivity from (5)

LHS $\Longrightarrow R \supseteq Q$ similarly

RHS $\Longrightarrow (R \cup P = R) \wedge (R \cup Q = R)$ definition of $\supseteq$

$\Longrightarrow (R \cup P) \cup (R \cup Q) = R \cup R$ adding the equations

$\Longrightarrow R \cup (P \cup Q) = R$ properties of $\cup$

$\Longrightarrow$ LHS definition of $\supseteq$

If $P \supseteq Q$, this means that Q is in all circumstances better than (or at least as good as) 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 <u>monotonic</u>, 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

$P \supseteq Q$	$\Longrightarrow P \cup Q = P$	definition $\supseteq$
	$\Longrightarrow F(P \cup Q) = F(P)$	property of =
	$= F(P) \cup F(Q)$	distrib F
	$\Longrightarrow F(P) \supseteq F(Q)$	definition $\supseteq$

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.

3.2 Least upper bounds

We have seen that $P \cup Q$ is the best program worse than both P and Q. Suppose now that we want a program better than both P and Q. In general, there will be no such program. Consider the two assignments

$$x := 1 \quad \text{and} \quad x := 2.$$

These programs are incompatible, and there is no program better for all purposes than both. If you want x to be 1, the second will be no good; whereas if you want it to be 2, the first program will be totally unsuitable.

Let us now 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 which is better than both, namely

$$x := 2$$

In fact there exists a worst program that is better than them both; and we will denote this by $P \wedge Q$

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

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

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

Corollary $P \sqsupseteq (P \wedge Q) \wedge Q \sqsupseteq (P \wedge Q)$ 3.1(1)

The operator $\cap$, wherever it is defined, is idempotent, symmetric and associative, and has unit $\perp$. Furthermore $\cap$ and $\not\triangleleft b \not\triangleright$ distribute through each other.

(2) $P \cap P = P$

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

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

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

(6) $P \cup (Q_1 \cap Q_2) = (P \cup Q_1) \cap (P \cup Q_2)$ provided that Q_1 and Q_2 are competible

(7) $(P \cap Q_1) \cup (P \cap Q_2) = P \cap (Q_1 \cup Q_2)$ provided that P and Q_1 are compatible, and P and Q_2 are are compatible

(8) $(Q_1 \cap Q_2) \not\triangleleft b \not\triangleright R = (Q_1 \not\triangleleft b \not\triangleright R) \cap (Q_2 \not\triangleleft b \not\triangleright R)$ provided that Q_1 and Q_2 are compatible

In the following laws we abbreviate $\bigcup_{i \leq n} x := E_i$ by $x :\in \{E_i \mid i \leq n\}$

Then we have

(9) $x := E$ and $x := F$ are compatible iff

$$\mathcal{D}E \wedge \mathcal{D}F \Longrightarrow E = F.$$

Furthermore in this case we have

$$(x := E) \cap (x := F) = (x := E \not\triangleleft \mathcal{D}E \not\triangleright x := F)$$

(10) $x :\in \{E_i \mid i \leq n\}$ and $x :\in \{F_j \mid j \leq m\}$ are compatible iff

$$\bigwedge_{i \leq n} \mathcal{D}E_i \wedge \bigwedge_{j \leq m} \mathcal{D}F_j \Longrightarrow (\{E_i \mid i \leq n\} \cap \{F_j \mid j \leq m\} \neq \emptyset).$$

Moreover in this case

$$(x :\in \{E_i \mid i \leq n\}) \cap (x :\in \{F_j \mid j \leq m\}) = x :\in (\{E_i \mid i \leq n\} \cap \{F_j \mid j \leq m\}) \not\triangleleft \bigwedge_{i \leq n} \mathcal{D}E_i \wedge \bigwedge_{j \leq m} \mathcal{D}F_j \not\triangleright$$
$$((x :\in \{E_i \mid i \leq n\}) \not\triangleleft \bigvee_{j \leq m} \overline{\mathcal{D}F_j} \not\triangleright (x :\in \{F_j \mid j \leq m\})).$$

(11) Assume that $b \Longrightarrow \mathcal{D}E$ and $c \Longrightarrow \mathcal{D}F$. Then $(x := E) \lhd b \rhd \bot$ and $(x := F) \lhd c \rhd \bot$ are compatible iff

$$(b \wedge c) \Longrightarrow (\{E\} \cap \{F\} \neq \emptyset).$$

In this case

$$((x := E) \lhd b \rhd \bot) \cap ((x := F) \lhd c \rhd \bot) = (x := E \lhd b \wedge c \rhd (x := E \lhd b \rhd (x := F \lhd c \rhd \bot))$$

(12) In general if $P = (x : \in \{E_i \mid i \leq n\}) \lhd b \rhd \bot$ and $Q = (x : \in \{F_j \mid j \leq m\}) \lhd c \rhd \bot$, they are compatible iff

$$(b \wedge c) \Longrightarrow (\{E_i \mid i \leq n\} \cap \{F_j \mid j \leq m\} \neq \emptyset)$$

and in this case

$$P \wedge Q = (x : \in (\{E_i \mid i \leq n\} \cap \{F_j \mid j \leq m\})) \lhd b \wedge c \rhd ((x : \in \{E_i \mid i \leq n\}) \lhd b \rhd ((x : \in \{F_j \mid j \leq m\}) \lhd c \rhd \bot))$$

The $\wedge$ operator generalises 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 $\bigcap S$

$$\bigcap S = P \wedge Q \wedge \ldots \wedge T \qquad \text{provided} \quad \exists R \;\; P \supseteq R \wedge Q \supseteq R \wedge \ldots \wedge T \supseteq R$$

It follows that

(13) $\qquad (\forall P \in S.\ P \supseteq R) \equiv \bigcap S \supseteq R \qquad$ provided $\bigcap S$ is defined.

This may be proved from law (1) by induction on the size of the set S.

If P and Q are compatible finite programs, then P ∧ Q can also be expressed as a finite program; indeed it can be reduced to normal form by use of the rules given above.

It is important to recognise that ∧ is not a combinator of our programming language, and that P ∧ Q is not strictly a program, even if P and Q are compatible programs. For example, let P be a program which assigns an arbitrary ascending sequence of numbers to an array, and let Q be a program which subjects the array to an arbitrary permutation. Then P ∧ Q would be a program that satisfies both these specifications, and consequently it would sort the array into ascending order. Unfortunately, programming is not so 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 just merge them into a single product satisfying all requirements. On the contrary, the engineer has to satisfy all requirements in a single design; and this is the main reason why designs and programs get complicated.

But no such problems arise with pure specifications, which may be freely connected by the humble conjunction "and". So P ∧ Q may be regarded as an <u>abstract</u> program, or specification of a program, that accomplishes whatever P accomplishes <u>and</u> whatever Q accomplishes, and doesn't fail except when <u>both</u> P and Q would fail. P ∧ Q (if it exists) specifies a program which is for all purposes better than both P and Q.

3.3 Limits

Now suppose S is a non-empty (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

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

Examples of directed sets are

$\{P\}$ a set with only one member

$\{P, P \sqcup Q\}$ since $P \sqcup Q \sqsupseteq P$ and $P \sqcup Q \sqsupseteq P \sqcup Q$

$\{P,Q,R\}$ where $P \sqsupseteq R \wedge Q \sqsupseteq R$

If S is finite and directed, then clearly it contains a member which is better than all the other members, and we have

$\bigsqcap S$ is defined

and $\bigsqcap S \in S$

If S is directed but infinite, then it does not necessarily contain a best member. Nevertheless, the set has a limit $\bigsqcap S$ which as before is the worst program better than all members of S. The set S is like a convergent series of numbers, which tends to a limit which is not a member of the series. By selecting members of the set it is possible to approximate arbitrarily close to its limit.

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.

$$(1)\quad (\sqcap S) \sqcup Q = \sqcap\{P \sqcup Q \mid P \in S\}$$

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

$$(3)\quad (\sqcap S);Q = \sqcap\{P;Q \mid P \in S\}$$

$$(4)\quad Q;(\sqcap S) = \sqcap\{Q;P \mid P \in S\}$$

$$(5)\quad b*(\sqcap S) = \sqcap\{b*P \mid P \in S\}$$

It is a fact about continuity 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

$$(6)\quad F(\sqcap S) = \sqcap\{F(X) \mid X \in S\}$$

3.4 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 $Q_0 = \bot$

$$Q_{n+1} = (P;Q_n) \lhd b \rhd \Pi \qquad \text{for all } n \geq 0$$

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 on the n^{th} iteration breaks, and can do anything ($\bot$). Clearly therefore

$$Q_n \geq Q_{n+1} \qquad \text{for all } n$$

(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

arbitrarily closely to the behaviour of the loop $(b*p)$. The loop itself can be defined as the limit of all its approximations

$$(1)\quad b*p \;=\; \bigcap\left\{Q_n \;\middle|\; n \geq 0\right\}$$

The same technique can be used to define a more general form of recursion. Let X stand for the name of the recursive program which we wish to construct, and let F(X) define the intended behaviour 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 behaviour of the recursive program

$$F^0(Q) \;=\; \bot$$

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

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

$$F^n(\bot) \geq F^{n+1}(\bot) \qquad \text{for all } n$$

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

$$(2)\quad \mu X.F(X) \;=\; \bigcap\left\{F^n(\bot) \;\middle|\; n \geq 0\right\}$$

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

$$(3)\quad b*p \;=\; \mu X.\ (p;X) \lhd b \rhd \mathrm{II}$$

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 law

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

Corollary $b^*p = (p;(b^*p)) \triangleleft b \triangleright II$

Proof

RHS $= F(\bigcap\{F^n(\bot) \mid n \geq 0\})$		by definition of μ
$= \bigcap\{F(F^n(\bot)) \mid n \geq 0\}$		by continuity of F
$= \bigcap(\{F^{n+1}(\bot) \mid n \geq 0\} \cup \{\bot\})$		definition of F^{n+1}, 3.2(5
$= \bigcap\{F^n(\bot) \mid n \geq 0\}$		since $F^0(\bot) = \bot$
$=$ LHS		dafinition.

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) $Y = F(Y) \Longrightarrow \mu X.F(X) \geq Y$

Proof.

$(Y = F(Y)) \Longrightarrow (\bot \geq Y) \wedge (Y = F(Y))$		
$\Longrightarrow (F(\bot) \geq F(Y)) \wedge (Y = F(Y))$		F monotonic
$\Longrightarrow (F(\bot) \geq Y)$		

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

$$Y = F(Y) \Longrightarrow F^{(n)}(\bot) \geq F^{(n)}(Y) \wedge Y = F^{(n)}(Y)$$
$$\Longrightarrow F^{(n)}(\bot) \geq Y$$

$$\Longrightarrow (\bigcap F^{(n)}(\bot)) \geq y \qquad 3.2(13)$$

as required.

4. Specifications

We have already in passing introduced two important concepts

(1) A specification or abstract program describes the intended behaviour of a program, but it is not itself a program because it is expressed in notations which are not permitted in the programming language

(2) A concrete program P may be <u>better</u> than an abstract program S; so whenever you want a program that behaves like S, the concrete program P will serve your purpose. In this case, we can say that P <u>satisfies</u> the specification S, or in symbols

$$S \supseteq P$$

It is the duty of the programmer, when given a specification S, to find a program P which satisfies S, and to prove that it does so. The practical purpose of the laws in this paper is to help in this task.

In this section we shall introduce 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. There is no reason to insist even that all specifications must be satisfiable. 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 $\mathbf{S}$ is <u>any</u> set of specifications, then

$\bigcap S$ is the specification which requires all R in S to be satisfied

$\bigcup S$ is the specification which requires some R in S to be satisfied.

The fact that these are limits of the sets is expressed

(1) $T \supseteq \bigcup S \equiv \forall R \in S.\ T \supseteq R$

(2) $\bigcap S \supseteq T \equiv \forall R \in S\ \ R \supseteq T$

Specifications obey without qualification all the laws of 3.2.

The $\supseteq$ ordering applies to specifications, just as it does to programs, but it can be interpreted in a new sense. If $S \supseteq T$, this means that S is a more general or weaker specification, and easier to meet than T: any program that satisfies T will serve for S, but maybe more programs will satisfy S. Thus $\bot$ is the easiest specification, satisfied by any program, and $\top$ is the most difficult (impossible, in fact).

4.1 Weakest prespecification

Abstract programs may be constructed in terms of all the operators available for concrete programs. For example S;T is a specification satisfied by a program that behaves like S; and when that terminates successfully, it behaves like T. 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 which meets the specification R. Perhaps we can think of a way to decompose this task into two simpler subtasks specified by S and T. The correctness of the decomposition can be proved by showing that

$$R \supseteq S;T$$

This proof should be completed before embarking on design for the subtasks S and T. Then similar methods can be used to find programs P and Q which

solve these subtasks, i.e., such that

$$S \supseteq P$$

and $$T \supseteq Q$$

It follows immediately from monotonicity of sequential composition that P;Q is a program that will solve the original task R, i.e.,

$$R \supseteq (P;Q)$$

Now suppose that in approaching the task R we can think of the second of the two subtasks T, but we do not know the first subtask. It would be useful simply to calculate S from T and R. We therefore define the weakest prespecification $T \backslash R$ to be specification which must be met by the first subprogram S in order that the composition (S;T) will accomplish the original task R. This fact is expressed in symbols

(1) $R \supseteq (T \backslash R);T$

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

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

$(x := 2 * x) \backslash (x := 4 * y) = (x := 2 * y)$

because $(x := 2 * y;\ x := 2 * x) = (x := 4 * y)$

$(x := 2 * x) \backslash (x := 3) = \top$

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

$(x := 2 * x) \backslash (x := 3 \vee x := 4) = (x := 2)$

because $(x := 3 \vee x := 4) \supseteq x := 4$

$= (x := 2;\ x := 2 * x)$

The law given above does not uniquely define $T \backslash R$. But of all the

solutions for X in the inequality

$$R \supseteq (X;T)$$

the solution $T \backslash R$ is the easiest to achieve. Thus if you want to find such a solution, a necessary and sufficient condition is that the solution should setisfy $T \backslash R$

$$(2) \quad R \supseteq (X;T) \;=\; (T \backslash R) \supseteq X$$

Thus in developing a sequential program to meet specification R, there is ro loss of generality in taking $T \backslash R$ as the specification of the left operand of sequential composition, given that T is the specification of the right operand. That is why it is called the weekest prespecification.

The specifiecation $P \backslash R$, where P is a program, plays a role very similar to Dijkstra's weakest precondition. It satisfies the analogue of several of his healthiness conditions.

In the following three laws, P must be a program.

(3) If you want to accomplish an impossible task, it is still impossible, even with the help of P

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

(4) If you want to accomplish twc tasks with the help of P, you must write a program that accomplishes ooth of them simultaneously

$$P \backslash (R1 \sqcap R2) = (P \backslash R1) \sqcap (P \backslash R2)$$

This distributive law extends to limits of arbitrary sets

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

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

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

Then $$P \setminus (\sqcup \mathbb{S}) = \sqcup \{P \setminus R_i \mid i \geq 0\}$$

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

(6) The program II changes nothing. Anything you want to achieve after II must be achieved before

$$II \setminus R = R$$

(7) If you want to achieve R with the aid of $P \vee Q$, you must achieve it with either of them

$$(P \vee Q) \setminus R = (P \setminus R) \wedge (Q \setminus R)$$

(8) If you want to achieve R with the aid of $(P;Q)$, you must achieve $(Q \setminus R)$ with the aid of P

$$(P;Q) \setminus R = P \setminus (Q \setminus R)$$

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

$$(P \lhd b \rhd Q) \setminus R = (P \setminus R) \lhd^{\check{}} b \rhd (Q \setminus R)$$

where $S \lhd^{\check{}} b \rhd T$ specifies a program as follows: if b is true <u>after</u> execution it has behaved in accordance with specification S, and if b is false afterwards, it has behaved in accordance with specification T. $P \lhd^{\check{}} b \rhd Q$ is not a program, even if P and Q are; in fact it may not even be implementable: consider the example

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

4.2 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 $/$, for example

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

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

(3) $\top/P = \top$ if P is a program

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

(5) $R/II = R$

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

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

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

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

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

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

Proof.

RHS	$= F(\bigcup \{P \mid R \supseteq F(P)\})$	definition F^{-1}
	$= \bigcup \{F(P) \mid R \supseteq F(P)\}$	F distributes
	$\subseteq R$	set theory

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

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

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

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

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

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

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

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

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

by (10) and the definition of $F^{-1}(P)$ we could conclude

$$P \supseteq F(P \cup Q)$$

which is false. The contradiction shows that F does not have an inverse, even in the weak sense described by (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, in secure knowledge that the final program F(X) will meet the original specification R

$$R \supseteq F(X)$$

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

5. Conclusion

The laws given in this paper are intended to assist programmers in reasoning effectively about their tasks, including both the development of programs that meet their specifications, and optimisation where necessary by algebraic transformation. The basic insight is that programs themselves, as well as their specifications, are mathematical expressions, and can therefore be used directly in mathematical reasoning in just the same way as expressions denoting familiar mathematical concepts such as numbers, sets, functions, groups, categories, etc. 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 stored-program digital computer.

The exposition of this paper is seriously incomplete in two important respects, one theoretical and one practical. The theoretical defect is that the laws are presented as self-evident axioms or postulates, intended to command assent from those who already understand what the laws are about. That is the way the laws of arithmetic or geometry are usually taught in schools. Nevertheless, as Russell points out, "The method of postulation has many advantages; they are the same as the advantages of theft over honest toil" [Russell]. Russell toiled hard to give a definition of the concept of a number in terms of more primitive concepts such as sets, and then to define the operations of arithmetic, and finally to prove that these definitions satisfy the laws that we never doubted in the first place.

If we were to embark on similar toil in the case of sequential programs and their specifications, the relevant mathematical definitions can be formulated within the classical theory of relations. This is done in a companion paper [Hoare and He]. The existence of such definitions, and their use to prove the laws enumerated in this paper, yields a valuable reassurance that the laws are consistent. Furthermore, it gives additional insight into the mathematics of programming, and how it may be applied in practice. In particular, it suggests additional useful laws; and it establishes that a given set of laws are <u>complete</u> in the sense that some wide and 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 a great comfort to the practising 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.

The second serious deficiency of the paper is the practical one. Even after nearly one hundred laws given in this paper, 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. The way ahead will be to gain practical experience in the application to programming of the kind of mathematics introduced in this paper, and to continue the search for deeper and more specific theorems which can be used more simply on limited but not too narrow ranges of problem. That is the way that applied mathematics, as well as pure mathematics, have made such great progress in the last two thousand years. If we follow that example, perhaps we may make faster progress, both in theoretical research and in its practical application.

References

J. Backus. Can Programming be liberated from the von Neuman style? Comm ACM 21.8 (1978) pp. 613 - 641

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

C.A.R. Hoare and He, Jifeng. Weakest Prespecifications. Technical Monograph PRG-44 (1985)

R.A. Kowalski. The relation between logic programming and logic specification. Mathematical Logic and Programming Languages. Prentice Hall (1985) pp. 11-27.

X.X. Murphy. Private Communication.

B. Russell. Introduction to Mathematical Philosophy. Allen and Unwin (1919).

D.S. Scott. Outline of a Mathematical Theory of Computation. Technical Monograph PRG-2 (1970).